
Universidade Estadual de Campinas
Instituto de Computação


Elder de Oliveira Rodrigues Júnior

# A Metamodel to Support the Formalization of Coding Conventions

# Um Metamodelo para Apoiar a Formalização de Convenções de Codificação

CAMPINAS
2020

**Elder de Oliveira Rodrigues Júnior**

# A Metamodel to Support the Formalization of Coding Conventions

# Um Metamodelo para Apoiar a Formalização de Convenções de Codificação

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

**Supervisor/Orientador: Prof. Dr. Leonardo Montecchi**

Este exemplar corresponde à versão final da Dissertação defendida por Elder de Oliveira Rodrigues Júnior e orientada pelo Prof. Dr. Leonardo Montecchi.

CAMPINAS
2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

R618m

Rodrigues Júnior, Elder de Oliveira, 1995-
A metamodel to support the formalization of coding conventions / Elder de Oliveira Rodrigues Júnior. – Campinas, SP : [s.n.], 2020.

Orientador: Leonardo Montecchi.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Convenções de codificação. 2. Análise estática. 3. Engenharia de software auxiliada por computador. 4. Programação orientada a objetos. 5. Geradores de código. I. Montecchi, Leonardo, 1982-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

**Título em outro idioma:** Um metamodelo para apoiar a formalização de convenções de codificação
**Palavras-chave em inglês:**
Coding conventions
Static analysis
Computer-aided software engineering
Object-oriented programming (Computer science)
Code generators
**Área de concentração:** Ciência da Computação
**Titulação:** Mestre em Ciência da Computação
**Banca examinadora:**
Leonardo Montecchi [Orientador]
Breno Bernard Nicolau de França
Daniel Lucrédio
Ricardo Terra Nunes Bueno Villela
**Data de defesa:** 13-08-2020
**Programa de Pós-Graduação:** Ciência da Computação

**Identificação e informações acadêmicas do(a) aluno(a)**
- ORCID do autor: https://orcid.org/0000-0003-4607-1632
- Currículo Lattes do autor: http://lattes.cnpq.br/1763542904122222


**Universidade Estadual de Campinas**
**Instituto de Computação**


**Elder de Oliveira Rodrigues Júnior**

# A Metamodel to Support the Formalization of Coding Conventions

# Um Metamodelo para Apoiar a Formalização de Convenções de Codificação

**Banca Examinadora:**

- Prof. Dr. Breno Bernard Nicolau de França
  Universidade Estadual de Campinas (UNICAMP)
- Prof. Dr. Daniel Lucrédio
  Universidade Federal de São Carlos (UFSCAR)
- Prof. Dr. Leonardo Montecchi
  Universidade Estadual de Campinas (UNICAMP)
- Prof. Dr. Ricardo Terra Nunes Bueno Villela
  Universidade Federal de Lavras (UFLA)

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 13 de agosto de 2020

# Agradecimentos

Ao **Prof. Dr. Leonardo Montecchi**, pela orientação excepcional, que sempre me ajudou pela busca de novos conhecimentos da área de pesquisa e me incentivou a participar do projeto internacional ADVANCE. Obrigado por acreditar em mim e pelos tantos elogios e incentivos.

Aos membros da banca examinadora, **Prof. Dr. Daniel Lucrédio**, **Prof. Dr. Breno Bernard Nicolau de França** e **Prof. Dr. Ricardo Terra Nunes Bueno Villela**, que tão gentilmente aceitaram participar e colaborar com a dissertação.

À minha **mãe** e ao meu **pai**, que sempre me apoiaram nesta jornada. À minha noiva e futura esposa **Sara**, por todo carinho e compreensão em momentos difíceis desta caminhada.

À **Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)**, processo nº 2018/11129-8 pelo financiamento do projeto de pesquisa, tornando possível a produção desta dissertação.

Por fim, a todos aqueles que contribuíram, direta ou indiretamente, para a realização desta dissertação, o meu sincero agradecimento.

# Resumo

As convenções de codificação são um meio de melhorar a confiabilidade dos sistemas de software. Elas podem ser estabelecidas por vários motivos, desde melhorar a legibilidade do código até evitar a introdução de falhas de segurança. No entanto, convenções de codificação geralmente vêm na forma de documentos textuais em linguagem natural, o que as torna difíceis de gerenciar e aplicar. Seguindo os princípios de engenharia orientados a modelos, nesta dissertação, propomos uma abordagem e uma linguagem para especificar convenções de codificação usando modelos estruturados. Chamamos tal linguagem de Coding Conventions Specification Language (CCSL). Também propomos uma transformação de modelo para gerar automaticamente verificadores a partir de uma especificação CCSL para encontrar violações das regras especificadas.

Para avaliar a proposta, realizamos dois experimentos. O primeiro experimento tem como objetivo avaliar o metamodelo CCSL, enquanto o outro tem como objetivo verificar a capacidade dos verificadores de encontrar violações da regra especificada nos códigos Java.

Os resultados obtidos são promissores e sugerem que a abordagem proposta é viável. No entanto, eles também destacam que muitos desafios ainda precisam ser superados. No primeiro experimento, analisamos um total de 216 regras individuais de dois grandes conjuntos de convenções de codificação existentes. No geral, foi possível representar 63% das regras de codificação consideradas usando nossa linguagem. No segundo experimento, selecionamos 53 regras dentre as implementadas na ferramenta PMD (um analisador de código popular) para comparar os resultados entre nossa ferramenta e a ferramenta PMD em três projetos reais. Em geral, alcançamos resultados iguais ou melhores da ferramenta PMD em mais da metade das regras selecionadas (79%), enquanto apenas 6% das regras não puderam ser especificadas usando nossa linguagem. Nas regras restantes, os resultados apresentados foram diferentes para cada uma das ferramentas.

Concluímos discutindo instruções para trabalhos futuros.

# Abstract

Coding conventions are a means to improve the reliability of software systems. They can be established for many reasons, ranging from improving the readability of code to avoiding the introduction of security flaws. However, coding conventions often come in the form of textual documents in natural language, which makes them hard to be managed and to enforced. Following model-driven engineering principles, in this dissertation we propose an approach and language for specifying coding conventions using structured models. We call this language Coding Conventions Specification Language (CCSL). We also propose a model transformation to concretely generate checkers to find violations of the rules specified with our language.

To evaluate the proposal, we performed two experiments. The first experiment aims to evaluate the Coding Conventions Specification Language metamodel, while the other aims to check the capability of the derived checkers to find violations of the specified rule in Java codes.

The obtained results are promising and suggest that the proposed approach is feasible. However, they also highlight that many challenges still need to be overcome. In the first experiment, we analyzed a total of 216 individual rules from two large sets of existing coding conventions. Overall, it was possible to represent 63% of the considered coding rules using our language. In the second experiment, we selected 53 rules from those implemented in the PMD tool to compare the results between our tool and the PMD tool in three real projects. In general, we achieve equal or better results of the PMD tool in more than half of the selected rules (79%), while only 6% of the rules could not be specified using our language. There are also cases where PMD performed better than our approach (9%) as well as cases where the results were different for each of the tools (6%).

We conclude by discussing directions for future works.

# List of Figures

# List of Tables

# Contents

# Chapter 1

# Introduction

Coding conventions [43], also termed as *coding standards*, are sets of guidelines for software development that recommend a certain programming style or specific practices, or impose constraints. Depending on their purpose, coding conventions may cover different aspects of software development, including file organization, indentation, comments, declarations, naming conventions, programming practices, programming principles, architectural best practices, etc.

Besides purely "cosmetic" recommendations, the adherence to precise coding rules is a fundamental practice for enforcing non-functional properties like security or performance. For example, attackers often exploit known vulnerabilities introduced by poor usage of programming constructs or system calls. Similarly, performance bottlenecks can be avoided by preferring certain programming constructs instead of others (e.g., see [49]). Coding conventions are not static artifacts; rather, they evolve over time following the introduction of new language features or the discovery of new vulnerabilities.

It has been argued that existing coding conventions — in their current shape — offer limited benefit, because of the difficulties in actually enforcing and managing them [19]. In fact, like many other artifacts in the development process, coding conventions mostly come in the form of textual documents written in natural language, possibly complemented with code examples. Thus, they cannot be processed automatically, and tasks like the following ones must be done manually: i) check similarity between rules, ii) identify conflicting rules, iii) understand if a tool is able to check a certain rule, iv) configure a tool to check a certain rule, etc.

In this dissertation, following the principles behind Model-Driven Engineering (MDE) [42], we investigate the possibility of specifying coding conventions through structured, machine-readable, models to increase the degree of automation of compliance of coding conventions. More in details, we provide the following contributions in this direction: i) we introduce a MDE-based approach for managing coding conventions as structured specifications; ii) we define a domain-specific language that realizes such approach for the Java programming language; iii) we run an experiment in which we use the defined language to specify existing coding conventions for the Java language; iv) we define a model-to-text transformation to concretely derive checkers from specifications of our language. v) we evaluate the transformation with real rules and real Java projects by comparing the results of the derived checkers to results of an existing static analysis tool.

## 1.1 Motivation

One of the motivations of this work stems from the MBSE initiative [45, 22], which aims to completely replace unstructured documents with structured models in the development of software and systems.

In fact, coding conventions usually come from a textual document, possibly with code examples. However, the time and effort required to verify if the development process is following the defined coding conventions may be very high. Static analysis tools exist, which can automatically check the conformance with coding conventions. Even so, most of those tools are developed to address specific coding conventions, and have limited extensibility. Also, understanding which tool can check which rules can be difficult, since there is a wealth of those tools, each one specific for a given programming language or (sub-)set of rules, and the documentation is also provided by textual documents.

We believe that formalizing coding conventions will, in the long run, open up the following possible scenarios:

i) Coding conventions could be automatically checked for compatibility, similarity, and conflicts between each other.

ii) Industrial standards could include formal definitions of imposed coding rules.

iii) Tool developers could expose, formally, the set of rules that their tool is capable to check. Traceability between rules and tools capable to check them would then be improved.

iv) Checkers could be automatically derived based on the formal specification of rules, using code generation techniques.

## 1.2 Publications

Regarding the publications of this Master's project, we have published the following paper:

- Towards a Structured Specification of Coding Conventions . The 24th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2019 (Pages 168 - 177). QUALIS B1.

In addition to this, the author of this dissertation participated in a technical visit for the period from 01/10/2019 to 31/01/2020 at the Università degli Studi di Firenze (UNIFI – Florence, Italy). The concrete objective of the exchange was to apply the methodology defined during the Master's project to the formalization of fault types, i.e., an approach to specify fault injections in Java source code. We have submitted the following paper which presents a model-driven approach to craft and inject software faults in Java source code.

- Model-Driven Fault Injection in Java Source Code . The 31st International Symposium on Software Reliability Engineering (ISSRE), 2020. QUALIS A2.

The above work has not been included in this dissertation, since it is a separate topic that only builds on the result of this project.

## 1.3 Dissertation Structure

The rest of this dissertation is organized as follows. In Chapter 2 we introduce the fundamental concepts to understand our approach. In Chapter 3 we discuss the related work. In Chapter 4 we present the overall approach together with its technical explanation, which is centered around a domain-specific language for specifying coding conventions. We define the CCSL language in Chapter 5, by discussing its metamodel (i.e., abstract syntax), together with usage examples. In Chapter 6, we demonstrate how we derive concrete checkers from the specification provided by our language. Still in Chapter 6, we also presents a prototype tool that implements our approach. The evaluation study, its results, and a discussion of the limitations are presented in Chapter 7. Finally, conclusions are drawn in Chapter 8.

# Chapter 2

# Fundamental Concepts

This Section presents the basic concepts on which the project is based, for a better understanding of its objectives.

## 2.1 Static Analysis

Verification & Validation (V&V) is a process that aims to ensure the quality of software [50], where: *validation* "is the process of evaluating software at the end of software development to ensure compliance with intended usage", and *verification* "is the process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase" [10, p. 11 ].

Static analysis is one of the techniques that are applied during V&V. The term "static" refers to the execution of techniques that do not involve the execution of the programs being analyzed [47]. Static analysis is valuable for discovering security problems caused by the implementation of software, since it does not take into account the execution of the program, but only its source code [8]. Static analysis of source code may be automated, e.g., searching for known erroneous coding patterns [25], which decreases the time and effort to find defects in source code.

Defining what to do to avoid introducing bugs is not an easy task, as relevant software defects tend to vary depending on the programming language (e.g., Java vs. C), on the domain (e.g., web services vs. embedded systems), and also on system requirements. For this reason, several *coding conventions* have been developed [19], which specify programming rules to be followed in order to avoid the introduction of defects, or more in general to improve some aspects of code quality. Coding conventions that are explicitly defined to produce high-quality code, and that are widely accepted in the community, are granted the status of *coding standard*. For example, MISRA C [26] and MISRA C++ [27] are coding standards for embedded systems developed in C/C++, and are widely used in safety-critical and mission-critical domains, e.g., automotive, railway, telecommunications, etc.

To a certain degree, tools exist that are able to check code for adherence to coding conventions. However the support is very fragmented, as most of the tools are specific for a given set of rules, or require complex implementation tasks for being extended. Further

details on existing tools are discussed in Chapter 3 (Related Work).

## 2.2 Model Driven Engineering

Model-Driven Engineering (MDE) [42] refers to the systematic use of models as primary artifacts throughout the engineering lifecycle. MDE techniques combine: i) Domain Specific Languages (DSLs) [52], which formalize the information relevant for a particular domain; and ii) model-transformations and generators [15], which analyze specific aspects of a model, and synthesize different kind of artifacts, such as source code, simulators, documentation, etc.

The ability to automatically transform models and synthesize various artifacts helps to ensure the consistency between system requirements, specification, implementation, and evaluation models. Furthermore, MDE reduces costs and human mistakes, by the application of state-of-the-art development practices, embedded in automated transformations. MDE techniques are widely used in practice in the industry and bring several benefits [21]. According to [54], the benefit of using such techniques is greater when applied to specific parts or aspects of the system. In fact, by targeting a specific problem, models and transformations can be even closer to the concepts of the domain.

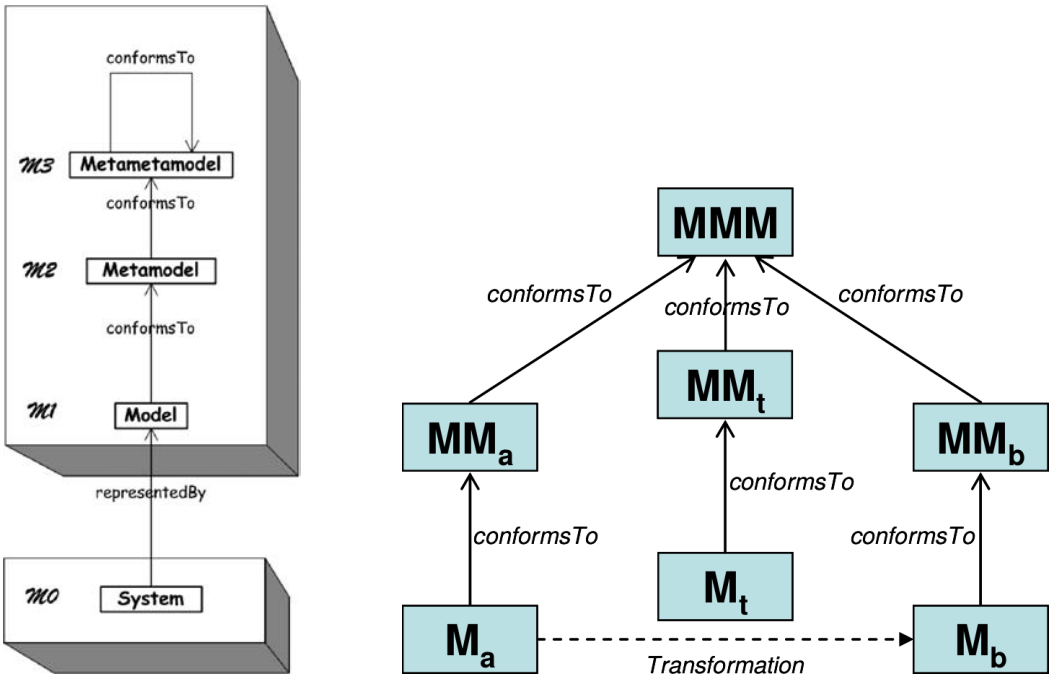
### 2.2.1 Foundations of MDE

One of the foundational concepts of MDE is metamodeling. A *metamodel* [14] formally defines what are the constructs that can appear in a certain class of models and their relations, that is, is a formalization of the abstract syntax of a language. A model is said to *conform to* a certain metamodel if it respects its abstract syntax. Following the "everything is a model" basic idea of MDE, metamodels can also be seen as models, expressed in a higher-level language called a *meta-meta-model*. In general, models are organized in "metamodeling layers" (Figure 2.1a) In practice, a metametamodel (M3) defines a language to define metamodels, that is, to define customized languages.

*Model transformation* is the process for which a model $M_b$, conforming to a metamodel $MM_b$, is automatically derived starting from a model $M_a$, conforming to a metamodel $MM_a$ (Figure 2.1b). This is typically done by executing a program (the *transformation*, $M_t$) written in some specialized transformation language (defined by metamodel $MM_t$) where rules are specified to map one metamodel with another.

The Object Management Group (OMG) is an international technology standards consortium focused on modeling; among the other things, it maintains the UML standard. OMG has defined its own comprehensive proposal for applying MDE practices: Model Driven Architecture (MDA). MDA is a set of standards that enable the derivation of value from models and software architecture to support full life cycle of physical, organizational, and computer systems [33]. The MDA set of standards addresses the representation and exchange of models, the transformation of models, the execution of models, and the generation of artifacts (e.g., code).

Among the standards provided by OMG, the Meta-Object Facility (MOF) [37] is of particular importance. MOF is an object-oriented meta-modeling language, that is, it




(a) [14] (b) [11]

Figure 2.1: Illustration of the concepts of metamodeling layers (a), and model transformation (b).

resides at level M3 in the hierarchy of Figure 2.1a. In practice, MOF is used to define Domain-Specific Languages (DSL), by formalizing their *metamodel* (abstract syntax). The UML itself is formally defined using MOF [38].

While MOF defines the abstract syntax, the Object Constraint Language (OCL) [34] can be use to define additional constraints that models should respect. Such constraints may involve complex relations between model elements, which would not be possible to represent using MOF, or whose representation would lead to an unnecessary complex language. OCL is also a query language, that is, can be used to obtain information on models.

Practical support for these concepts exist in different tools. One of the most complete proposals is the Eclipse Modeling Framework (EMF) [48], which provides a Java-based implementation of MOF known as Ecore.

### 2.2.2 Applications of MDE

There exist a large spectrum of applications for MDE techniques, which go beyond basic code generation [54]. This project applies MDE techniques to improve the management of coding conventions, and it is especially related to two applications of MDE: *Model-Based System Engineering* and *Model-Driven Reverse Engineering*, described in the following.

**Model-Based Systems Engineering.** The Model-Based Systems Engineering (MBSE) initiative, promoted by the INCOSE consortium [45, 22] in the field of systems engineering, has the objective to simplify the management of information involved in the design,

development, and assessment of complex systems. In the MBSE philosophy, *all* the artifacts in the development of a software or system should be managed as models. The ultimate objective is to switch from a document-centric process to a model-centric process, in order to improve traceability and synchronization of artifacts addressing different aspect of a complex system. Our proposal also goes in this direction, supporting the transition of coding conventions from textual document to structured models.

**Model-Driven Reverse Engineering** Model Driven Reverse Engineering (MDRE) can be seen as the application of MDE principles and techniques to reverse engineering [13], that is, any information extracted from source code will be organized in models that conform to precise metamodels [40]. MDRE approaches contribute to maintenance and evolution processes of the software providing models able to give compact information about thousand lines of code. Queries on such models, e.g., specified using OCL, can be applied to easily retrieve valuable information and metrics on the software under analysis.

Furthermore, since the source code "becomes" a model, MDE techniques can be applied on it, e.g., model-to-model transformations can be applied to refactor the architecture or specific implementation aspects, and code generation can be later used to (re-)generate the code of the updated system.

MDRE approaches extract models from source code, while in this project we aim at *creating models of coding rules*, that is, formalizing constraints on source code. The two aspects are obviously interrelated. Understanding to what extent the work made in the reverse engineering field can be reused is also an aspect of the proposed work.

## 2.3 Coding Conventions

Frequently, even in the scientific literature and in tool manuals, the terms "coding convention", "coding standard", "coding rules", etc., are used interchangeably. To avoid ambiguity, we give here a brief definition of these terms for the context of this project.

A *coding rule* is a restriction on the possible ways to program software; it states the conditions under which a portion of code must be considered invalid for the purpose of a software project. More formally, considering $\mathcal{C}$ as the set of all the possible portions of source code, a coding rule is a function $f\colon \mathcal{C} \to \{\texttt{valid}, \texttt{invalid}\}$. Some rules only concern formatting aspects, e.g., naming of variables or placement of brackets. Enforcing such rules does not require altering software behavior. We call this particular class of rules *coding style rules*.

A *coding convention* is a set of coding rules, usually having a specific purpose, e.g., improving security or performance. Many coding conventions are created for the purpose of a single project or company, and they never reach the public domain. Conversely, we consider a coding convention to be a *coding standard* when it is widely recognized in its reference community, or when it is actually published as a technical standard (e.g., MISRA C++ [27] or the JPL Java Coding Standard [23]).

In current practice, a wealth of coding conventions exists. For example, in the study in [44], an interview among 7 software engineers about the most important practices for

software maintainability resulted in 71 different coding rules, and different opinions on their relative priority.

The Software Engineering Institute (SEI) of Carnegie Mellon University provides a set of coding conventions focused on security problems [46], which supports the development using popular languages, such as Java and C++. Each coding convention provided by SEI contains rules where each rule is organized according to the following dimensions: *severity* (low, medium, and high), *likelihood* (unlikely, probable, likely), *remediation cost* (low, medium, and high), *priority* (P1 to P27), and *level* (L1 to L3). Figure 2.2 [46] illustrates how the features of the code convention are related to each other.

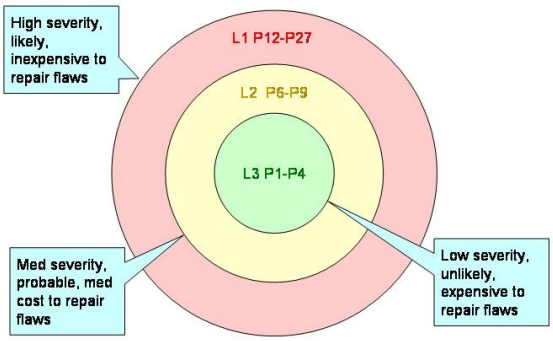



Figure 2.2: Classification of coding conventions provided by SEI

Many companies define their own coding conventions, which may be different among different teams or even for individual developers. This can happen, for example, because of different programming languages, of different project requirements, or simply because a certain client imposes its specific restrictions. Typically, coding rules are specified using the natural language. Sometimes they are coupled with code examples, to demonstrate the addressed problem and how enforcing the rule would remove it.

The author of [19], scientist at NASA/JPL, argued that the benefit of existing coding conventions is often small, even for critical applications. The main reason for such lack of effectiveness was attributed to the lack of comprehensive tool-based (i.e., automated) compliance checks. Indeed, while the support of automated tool has improved in recent years, it can not be said that comprehensive automated compliance checks are common. Tool support is fragmented: each static analysis tool checks a different set of rules, often for a specific programming language. Verifying all the rules of a certain coding convention needs the combined application of multiple tools, if they exist at all. This is especially true when customized conventions need to be enforced. More importantly, some rules are too generic, making it hard to implement or even to model it. For example, some rules in the SEI CERT coding conventions go into the details of the unpacking of compressed files (IDS04-J), while other ones simply give the generic recommendation to "perform proper cleanup at program termination" (FIO14-J).

## 2.4 MDE Tools

The following sections present fundamental concepts and examples about the MDE tools that we used in this study. In particular, our project is based on EMF (2.4.1), OCL (2.4.2), Acceleo (2.4.3), and MoDisco (2.4.4).

### 2.4.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [48] is a well-known meta-modeling framework that provides an unifying runtime layer for different MDE tools within Eclipse.

The EMF defines a dedicated meta-metamodel (Ecore) to define new metamodels and models. In fact, the Ecore language is an implementation of the MOF language (Meta-Object Facility) specified by the OMG. To better understand how languages are defined in Ecore, consider a domain specific language to describe a research group where it should be described its members, its publications, and its collaborators (people who collaborated to some publication but are not members of the group). The Figure 2.3 illustrates the formalization of a metamodel using Ecore to represent such a domain.

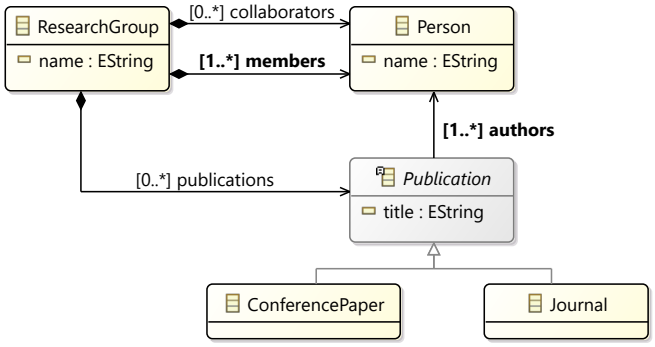



Figure 2.3: Ecore metamodel of the Reseach Group domain

The EMF provides the developers with different means to create a metamodel (e.g., via XML schema or UML diagrams). As it can be seen in Figure 2.3, the metamodel is represented via UML class diagram, as we believe that this format makes clear the visualization of which classes and their relationships exist in the metamodel. We formalized the research group domain by creating the following metaclasses:

1. **ResearchGroup**. It represents a research group. A ResearchGroup is composed of: i) a name; ii) members, as a list of *Person*; iii) publications, as a list of *Publication*; and iv) collaborators, as a list of *Person*.

2. **Person**. It represents a person who is a member or a collaborator of a research group. A Person contains a name.

3. **Publication**. It represents a publication. A Publication contains a title and a list of its authors. A Publication is an abstract metaclass, and it can be extended by concrete metaclasses. For the sake of simplification, we are considering only two kind of publications (Conference papers and Journals).

While the metamodel describes the structure of the model, a model is a concrete instance of the metamodel. Defining a metamodel using the Ecore allow us to take advantage of the entire environment of the Eclipse Modeling Framework (EMF). Among the many features provided by the EMF, the code generation is particularly relevant. From the Ecore metamodel, we can automatically generate a Java implementation that represents the specified domain. By generating the Java implementation of the metamodel, we can create an instance of the metamodel in different ways. For example, consider a model of a research group named as 'LASER' with two members ('Elder Rodrigues' and 'Leonardo Montecchi'), and with one collaborator ('Ricardo Terra') and with two publications (an conference paper with title 'Towards a Structured Specification of Coding Conventions' written by 'Elder Rodrigues' and 'Leonardo Montecchi' , and a journal with title 'How Do Developers Use Dynamic Features? The Case of Ruby' written by 'Elder Rodrigues' and 'Ricardo Terra'). The described instance of the metamodel can be achieved in different means:

1. Using the Generated Model APIs. A model can be created using the generated Java implementation as illustrated in Figure 2.4. The EMF also provides an API to serialize the model in a XMI format.

```
1  //Registry the domain model
2  EPackage.Registry.INSTANCE.put(ResearchgroupPackage.eNS_URI,
       ResearchgroupPackage.eINSTANCE);
3
4  //Creates LASER Research Group
5  ResearchGroup laser = ResearchgroupFactory.eINSTANCE.createResearchGroup();
6  laser.setName("LASER");
7
8  //Add LASER member - Elder
9  Person elder = ResearchgroupFactory.eINSTANCE.createPerson();
10 elder.setName("Elder Rodrigues");
11 laser.getMembers().add(elder);
12
13 //Add LASER Member - Leonardo
14 Person leonardo = ResearchgroupFactory.eINSTANCE.createPerson();
15 leonardo.setName("Leonardo Montechi");
16 laser.getMembers().add(leonardo);
17
18 //Add LASER Collaborator - Ricardo
19 Person ricardo = ResearchgroupFactory.eINSTANCE.createPerson();
20 ricardo.setName("Ricardo Terra");
21 laser.getCollaborators().add(ricardo);
22
23 //Add LASER Conference Paper Publication
24 ConferencePaper confPaper = ResearchgroupFactory.eINSTANCE.createConferencePaper();
25 confPaper.setTitle("Towards a Structured Specificationof Coding Conventions");
26 confPaper.getAuthors().add(elder);
27 confPaper.getAuthors().add(leonardo);
28 laser.getPublications().add(confPaper);
29
30 //Add LASER Journal Publication
31 Journal journal = ResearchgroupFactory.eINSTANCE.createJournal();
32 journal.setTitle("How Do Developers Use Dynamic Features? The Case of Ruby");
33 journal.getAuthors().add(elder);
34 journal.getAuthors().add(ricardo);
35 laser.getPublications().add(journal);
```

Figure 2.4: An instantiation of the Research Group domain using the generated Java code

2. Using the default EMF Editor. The EMF also generates an entire Eclipse plug-in dedicated to create models of the metamodel via a tree editor, which respects the

constraints imposed by the metamodel. However, depending on the metamodel this alternative may not be convenient, as it is not clear to see the relations between the instances using this editor. Figure 2.5 illustrates an instance of the research group metamodel.

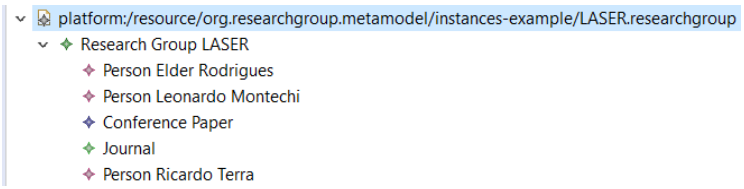



Figure 2.5: An instantiation of the Research Group domain using the generated editor

3. Creating a textual syntax. For practical reasons, sometimes a textual syntax of the metamodel works better than other alternatives. There are also tools which aid to create textual syntax of metamodels and provide mappings to parse the textual instantiation to the Ecore objects (e.g., XText [7]). The Figure 2.6 illustrates an instantiation of the research group domain using a concrete textual syntax.

```
1  lab LASER;
2
3  members: {
4    'Elder Rodrigues',
5    'Leonardo Montecchi'
6  }
7
8  collaborators: {
9    'Ricardo Terra'
10 }
11
12 publications: {
13   Conference-Paper: 'Towards a Structured Specification of Coding Conventions'
         written by 'Elder Rodrigues', 'Leonardo Montecchi',
14   Journal: 'How Do Developers Use Dynamic Features? The Case of Ruby' written
         by 'Elder Rodrigues', 'Ricardo Terra'
15 }
```

Figure 2.6: An instantiation of the Research Group domain according to a textual syntax

4. Using a graphical editor. Finally, another suitable alternative is to create graphical views of the model as illustrated in Figure 2.7. Again, there are also tools which aid to create graphical syntax of metamodels and provide mapping to parse the graphical instance to the Ecore objects (e.g., GMF [3] and Sirius [6]). In particular, we use the notation like in Figure 2.7 in the rest of this dissertation, which is based on the Object Diagram of UML.

### 2.4.2 OCL

The metamodel formally describes the structures and relationships of a domain. However, domains often contain complex rules that can not be defined through the definition of the structures and relationships. To support such rules, we can use the Object Constraint Language (OCL) [34], which is a declarative language used to specify constraints in models. The EMF also provides an implementation of the OCL to be applied in Ecore models.

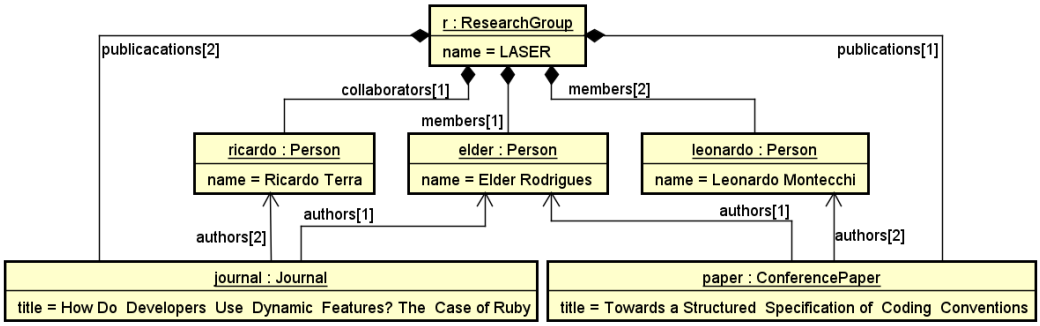



Figure 2.7: Graphical view of an instantiation of the Research Group domain

Still considering the previous example of the research group domain, suppose that every name of a person must begin with a capital letter. Such a rule can be validated using the OCL illustrated in Figure 2.8.

```
1 context Person
2 inv: if self.name.size() > 0 then
3   self.name.at(1) = self.name.at(1).toUpperCase()
4 else
5   true
6 endif
```

Figure 2.8: An OCL used to define constraint in models

Besides the original purpose of OCL, it also can be used as a query language to retrieve model elements according to some criteria. For example, the following OCL selects all publications written by the authors 'Elder Rodrigues' and 'Leonardo Montecchi'.

```
1 Publication.allInstances()→select(p: Publication |
2   p.authors→exists(e: Person | e.name = 'Elder Rodrigues') and
3   p.authors→exists(l: Person | l.name = 'Leonardo Montecchi')
4 )
```

Figure 2.9: An OCL to retrieve all publications written by 'Elder Rodrigues' and 'Leonardo Montecchi'

### 2.4.3 Acceleo (Model-to-Text Transformation)

Model-to-Text transformation is the generation of textual artifacts from models. Implementing model-to-text transformation using general languages such as Java may create too much verbose code or even redundant codes. Nowadays, there are dedicated frameworks and tools to facilitate the definition of model-to-text transformation. In this work, we used the Acceleo language, which is a template-based model-to-text technology that follows the standards defined by the OMG (Object Managament Group), i.e., it implements the MOFM2T (MOF model-to-text) standard [36]. Acceleo supports Ecore models, which means that any metamodel described using the Ecore models could be used in the language.

To better understand how Acceleo works in practice, Figure 2.10 illustrates a fragment of an Acceleo code that generates a simple HTML file taking as input an instance of the

research group metamodel.

```
1 [template private writePublications(researchGroup: ResearchGroup)]
2 <ul>
3   [for(p: Publication | researchGroup.publications) separator('\n')]
4   <li>([p.writePublicationType() /]) [p.title /] ([p.writeAuthors() /])</li>[/for]
5 </ul>
6 [/template]
```

Figure 2.10: Model-to-text transformation using Acceleo to generate a web page for a Resourch Group instance

*Templates* are blocks that describe the text being generated in a pre-formatted way. The first parameter of a template indicates the type allowed to invoke the template (e.g., only objects of type ResearchGroup can invoke the writePublications template). Templates can be grouped into *Modules*. Similarly to the most object oriented languages, Acceleo also supports controlling access of templates through a visibility mechanism (e.g., private and public).

By passing the LASER instance of the research group metamodel as input to the above Acceleo code, the HTML file illustrated in Figure 2.11 is generated.

```
1  <html>
2    <title>LASER</title>
3    <body>
4      LASER Members:
5      <ul>
6        <li>Elder Rodrigues</li>
7        <li>Leonardo Montecchi</li>
8      </ul>
9      LASER Collaborators:
10     <ul>
11       <li>Ricardo Terra</li>
12     </ul>
13     LASER Publications:
14     <ul>
15       <li>(Conference Paper) Towards a Structured Specificationof Coding Conventions
             (Elder Rodrigues, Leonardo Montecchi)</li>
16       <li>(Journal) How Do Developers Use Dynamic Features? The Case of Ruby (Elder
             Rodrigues, Ricardo Terra)</li>
17     </ul>
18   </body>
19 </html>
```

Figure 2.11: LASER.html file generated by the Acceleo transformation

### 2.4.4 MoDisco

The MoDisco tool is an open source Model Driven Reverse Engineering framework (MDRE), which aims to provide model representation from any kind of possible system artifacts as illustrated in Figure 2.12 [13].

Regarding the Java technologies implemented in MoDisco, the developers also defined a metamodel of the Java Language using the Ecore metamodeling language to precisely represent Java codes, i.e., a one-to-one representation of the source code. Following the MDRE (Model Driven Reverse Engineering) approach, there is also a extractor which extracts a structured model of the Java source code which conforms to the defined Ecore

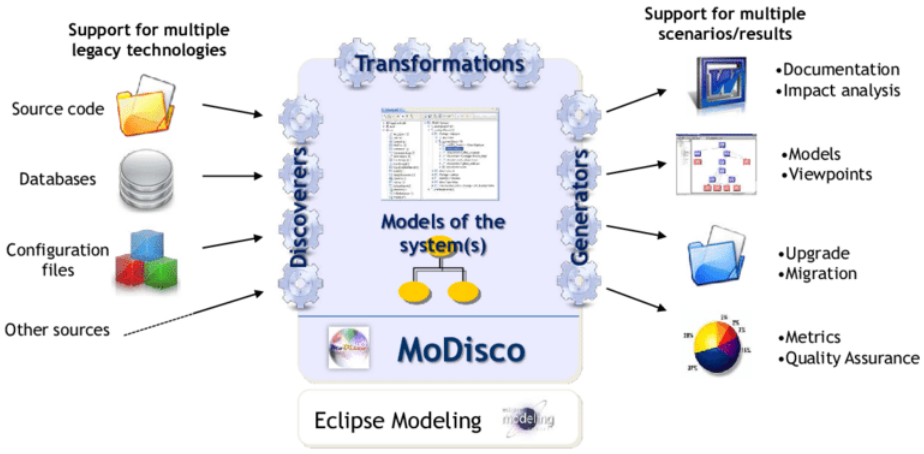


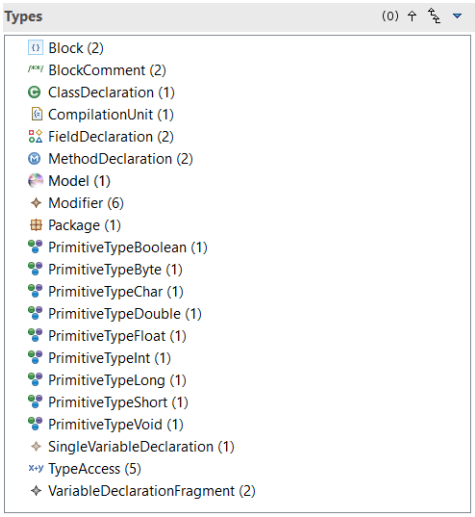

Figure 2.12: Overview of the MoDisco project [13]

Java metamodel, as well as a automated regeneration of the Java source code from the extracted model. The MoDisco also provides model-to-model transformations which transform a Ecore Java model to a Ecore KDM model, i.e., a model to represent source code but which is not bounded to the Java language. However, as our approach is specifically to the Java language, we use the generated Ecore Java Model.

The Figure 2.13 illustrates the Java model discovered by MoDisco (Figure 2.13b) of a Java code (Figure 2.13a).

```java
package sample;

public class Sample {

    private int bar;
    private boolean quuz;

    public void foo(int x) {/* code */}

    public static void qux() {/* code */}
}
```

(a)




(b)

Figure 2.13: An example of a Java model discovered by MoDisco

# Chapter 3

# Related Work

## 3.1 Static Analysis Tools

A straightforward means to verify adherence to coding standards is to perform manual code review. This is of course a very expensive process. Over the years, tools to automate the verification of rules on source code have emerged. Typically, they are based on *static code analysis*, which consists in searching source code for common defects and known bug patterns, without executing the software itself. Several tools exist for this purpose.

One of the first tools targeting the Java language was FindBugs (now SpotBugs) [20], which was originally created to detect null pointer defects. It has then evolved with the support of additional rules, and it features a plugin module that can be used to write customized detectors for additional bug patterns. Similarly, QJ Pro [5] checks conformance to a predefined set of formatting rules, misuses of the Java language, code structure, etc. Unfortunately, from the available documentation we were not able to precisely determine which rules are supported by this tool. The development of QJ Pro seems to have stopped several years ago.

Many other tools for static code analysis exist. While most of them provide some kind of extension mechanism, adding or modifying rules is typically a complex task, which requires low-level manipulation of the abstract syntax tree of the code under analysis. SonarSource techonologies (SonarQube and SonarLint), PMD [4] and CheckStyle [2] are three of the most configurable tools for Java. Both of them can analyze code for compliance with either predefined rules, or rules created by users using scripts or configuration languages. Typically, customized checks are defined using the APIs provided by the tool, which basically consists in implementing a visitor pattern on the abstract syntax tree using Java code. Depending on the language being analyzed, such tools may also support the definition of rules through XPath queries [55] on a XML-based representation of the AST[1]. XPath is a query language for XML documents; beside being very verbose, the developer has to explicitly take into account for every possible form of appearance of the violation. For example, the rule *DontCallThreadRun* defined in the PMD tool is defined using XPath as illustrated in Figure 3.1. However, such a specification contains many low level concepts.

[1] `https://pmd.github.io/pmd/pmd_userdocs_extending_writing_rules_intro.html`

```
1  //StatementExpression/PrimaryExpression
2  [
3      PrimaryPrefix
4      [pmd-java:typeIs('java.lang.Thread')]
5      [
6        ./Name[ends-with(@Image, '.run') or @Image = 'run']
7        or
8        ./AllocationExpression/ClassOrInterfaceType[pmd-java:typeIs('java.lang.Thread')]
9              and ../PrimarySuffix[@Image = 'run']
10     ]
11 ]
```

Figure 3.1: *CallSuperFirst* PMD implementation

The Sonar analyzers for Java also construct a semantic model of the code. The semantic model provides information related to each node of the abstract syntax tree being manipulated, e.g., the variable's owner or all super types of a class/interface. With such feature, it is also possible to create rules related to architectural design, e.g., only classes annotated with "@Service" can call methods of a class annotated with "@Repository".

A study comparing FindBugs, QJ Pro, and PMD can be found in [53], while a more general survey on static analysis techniques and tools can be found in [18].

The authors of [17] define an approach to provide machine-readable specifications of coding rules for CSS. Another interesting approach has been introduced in [9] and implemented in Naturalize, a tool based on Natural Language Processing (NLP), which analyzes a code base to first recognize naming and formatting conventions adopted in the project, and then identify possible violations. However, these approaches only address formatting and naming issues (i.e., coding style), and there is no way to specify customized coding rules that address security aspects, for example. There is however a growing trend of applying machine learning techniques to static analysis. For example, the authors of [39] use algorithms based on decision trees to identify violations to coding style rules for Java.

The work in [31] benchmarks different static analysis tools with respect to their ability to identify vulnerabilities. The results highlighted that the best solution depends on the deployment scenario and class of vulnerability being detected. In [32], the same authors show that combining multiple static analysis tools does not necessariliy improve the results over using a single tool. The authors of [56] focus on structuring the relations between rules and vulnerabilities across different repositories. However, they do not provide a structured specification of the rules themselves.

In contrast to the presented static analysis tool, in this work we define a generic tool-agnostic language to specify coding conventions, from which low-level tool-specific configurations can be derived by model transformation. For example, Sonar analyzers or PMD analyzers can derived through a specification of our language.

## 3.2 Modeling of Source Code

A survey of MDE techniques for reverse engineering can be found in [40]. In this respect, the Knowledge Discovery Metamodel (KDM) [35], defined by the Object Management

Group (OMG), is of particular relevance. KDM is a metamodel for representing existing software: it considers the physical and logical elements of software at various levels of abstraction, as well as their relations. The primary purpose of this metamodel is to enable a common interchange format for interoperability between tools, as a vendor-neutral format. MoDisco (Model Discovery) [13] provides a concrete implementation of the metamodel, and it supports the extraction of KDM models from software. However, the objective of KDM is to model an entire software project in its details, while our objective is to model coding conventions at a higher level of abstraction.

The QL language [12] is a query language that has been mostly applied to the specification of queries on source code. QL is considered a general-purpose query language [12], while our objective is to define a domain-specific language for the specification of coding conventions. Also, because it supports arbitrary queries on source code, QL is necessarily quite verbose, while we aim at a concise specialized language. Finally, we are proposing a complete MDE workflow, in which our metamodel is the basis for deriving other artifacts (see Figure 4.1 earlier). QL queries could be an example of derived artifacts, as discussed in the next section.

Other works in the literature focused on modeling different aspects of source code. Some of them focused on the formalization of so-called code smells (e.g., [28]), which however are only one of the reasons that drive the definition of coding conventions. In fact, code smells are also ensured by static analyzers, e.g., the Sonar analyzers contains many automated code smells implementation. There are few works which contributes to the detection of code smells in a way similar to our contribution to detect violations of coding conventions, i.e., providing a model driven technique to select coding portions of the source code.

The work in [24] also uses OCL queries to select codes portion considered as code smell. The OCL queries are performed in the JavaEAST model, which is a model representation of the source code created by them. However, the OCL queries should be created manually, which is not the intention of this work. The authors of [29, 30] defined a Domain-Specific language to specify coding smells. They formalize the specification of code smells with a BNF (Backus–Naur form) grammar. In their metamodel, the modeler can specify a code smell based on the lexical names (e.g., class names), metrics (e.g., number of lines of a specific class), and relationships (e.g., classes that references another classes). It is also possible to combine multiple rules through a set of operators. For example, using their metamodel we can create a definition of "Spaghetti code" with a composition of rules that detects a class that uses global variables where it also declares many methods with no parameters and a higher number of lines. Although their work is quite related to our work, we are interested to a more precise relation of source code elements since our goal is to define coding conventions (e.g., classes that define the clone method should also implements the Cloneable interface).

# Chapter 4

# Proposed Approach

## 4.1 General Workflow

Our workflow for the management of coding conventions as structured, machine-readable, specifications is depicted in Figure 4.1. The workflow is centered around a domain-specific language that is used to provide such structured specifications. We call this language the Coding Conventions Specification Language (CCSL). Textual version of existing coding rules are translated into specifications in such language, while new rules can be created directly as CCSL models.

Once the machine-readable specification of coding rules is available, they can be analyzed, for example to identify conflicting rules or equivalent ones. Then, from such specifications, model-transformations can be applied to automatically derive different artifacts.

Being able to specify coding rules through a domain-specific language allow us to focus on the context of the rule rather than how should we implement it. For example, in order to extend a rule in the existing static analysis tool, it is required some tasks which is subject to error-prone: i) how to traverse the AST (abstract syntax tree); ii) which AST elements matches to the subject rule elements; and iii) how AST elements are related to each other. In this context, deriving checkers to perform such tasks gives to the developer an abstraction level which could facilitate extending and introducing new rules. We identify three classes of artifacts that can be derived from CCSL models.

First, there are some kind of artifacts that are not related with static analysis that could be derived. For example, an explanation of the rule in natural language, or examples of source code portions that violate it.

Second, it is possible to derive specific implementation for existing static analysis tools using their APIs. Several tools for static code analysis provide APIs in order to implement new custom rules, although it does not mean that defining custom rules is a trivial task. In general, static analysis tools extract a model representation of the source code (e.g., Abstract Syntax Trees) and inspect into it to find violations.

Third, there may be situations in which existing static analysis tools could not be used to perform some checks due to some technical limitation. In this case, the specified coding rules can be verified by deriving Java codes able to check the rule without any dependency on the target static analysis tool. Although this approach is the most generic

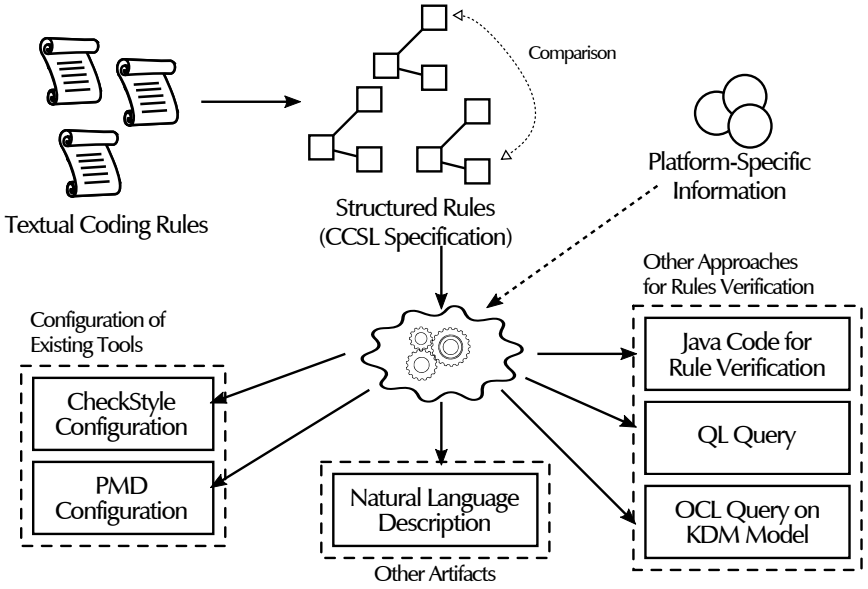



Figure 4.1: The proposed workflow for a the management of coding conventions.

possible, it is also quite complex to be concretely implemented.

Finally, we would like to highlight the last approach that is what we have fully implemented in this work. By considering the MDE context and the capabilities of the available tools related to MDE, another alternative is to derive OCL queries. In this approach we use the MoDisco tool, which is able to extract a Java model from the Java code, i.e., it provides a one-to-one representation of the source code elements. We then execute OCL queries in the extracted Java Model where the retrieved elements are violations of some rule. The OCL queries are generated automatically thanks to our implementation of a model-to-text transformation where the CCSL specification is the input. To a better understanding in how we implemented this approach, the next Section (Technical Workflow) describes in details the defined workflow and its related tools.

## 4.2 Technical Work Flow

To better understand how we concretely implemented the proposed workflow, we provide here more details about the artifacts generated.

The detailed workflow is presented in Figure 4.2 and discussed in the following.

1. **Rule Specification:** This step consists in writing a rule specification, i.e., to create the structured model using the CCSL metamodel.

2. **Checker Generation:** In this step an OCL query is automatically generated. As previous mentioned, the generated checker (OCL query) also adopts a model-driven approach, that is, the manipulations are made on a structured representation of the code and not on the code itself. In other words, the OCL manipulates the Java models extracted by the Modisco Tool. The elements retrieved by the OCL query are violations of the rule specified in the previous step.

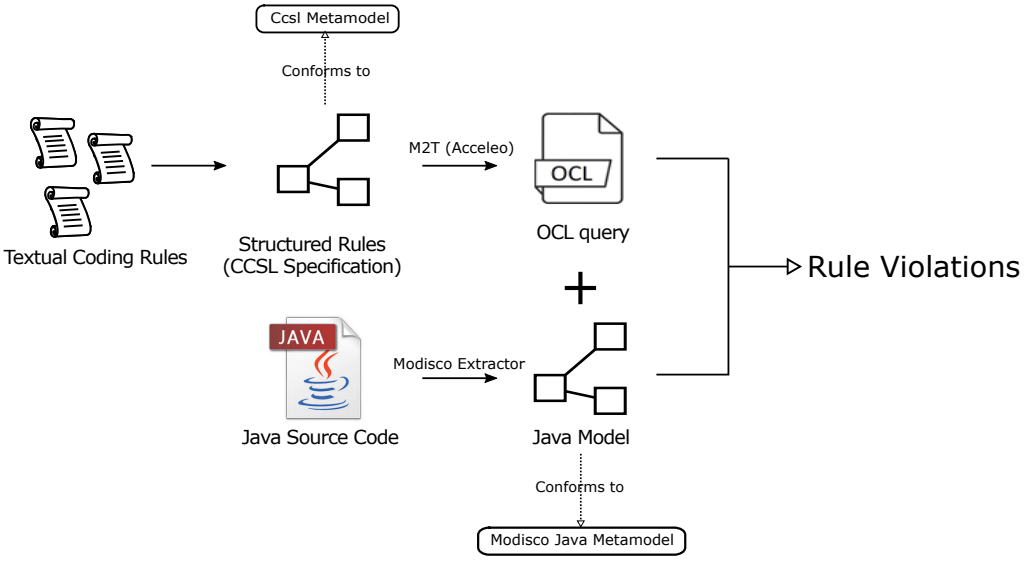



Figure 4.2: The concrete workflow that we applied to implement the proposal in Figure 4.1

3. **Source Code Extraction:** In order to execute the OCL query generated in the previous step, the structured model of the target source code must be available. To perform such task we rely on the MoDisco tool [13], whose purpose is exactly to extract a detailed structured model from a Java project.

4. **Violations Identification:** Once the structured model of the target project is available, the generated OCL query is executed on it. The set of the violations is the output of this step. Here the the corresponding file and line of the code elements identified as violations are printed.

The CCSL elements and the transformations are detailed in Chapter 5 (CCSL Implementation).

## 4.3 Main Challenges

To realize the workflow described in the previous section, several technical and research challenges need to be overcome. We highlight the most important ones in the following.

### 4.3.1 Vocabulary

The first challenge consists in the vocabulary to be considered, since different programming languages may use different terms to represent the same concept. To minimize this problem, in this paper we focus on the Java language only. Extension to other programming languages will be investigated as future work.

### 4.3.2 Abstraction Level

A model is an abstraction of reality, in the sense that it cannot represent all aspects of reality [14]. When trying to define a structured representation of coding rules, we

are trying to define *models* of such rules. Such models are expressed in CCSL, whose abstract syntax is defined by a *metamodel*. Finding the appropriate abstraction level of the metamodel is challenging, especially because the notion of preciseness of a model is not an absolute notion [14].

While a lower abstraction level allows more coding conventions to be specified, it results in a complex and verbose metamodel, resembling the abstract syntax of the programming language itself, and thus going in the opposite direction with respect to a domain-specific language. On the other hand, a higher abstraction level simplifies the specification of coding rules for the user, but it makes the derivation of low-level artifacts more difficult. We base our metamodel mainly on the Modisco's Java metamodel, which is a 1:1 representation of the source code. Due to that the Modisco's Java metamodel is not appropriate to specify coding conventions, since it has too many low level elements. Taking inspiration from it, we have a model focused on specifying coding rules. For all the abstraction we made, we made sure that it is possible to generate an analyzer that actually identifies the violations of the specification in the source code code.

### 4.3.3 Rules Verification

Even if a coding rule can be specified using CCSL, this does not guarantee that it may be easily verified using some static analysis tool or program code. In fact, some rules may be hard or impossible to be verified using static analysis only. For example, performance bottleneck rules that involve complex analyzes of parallel tasks can be difficult to detect with static analysis alone, since they can throw many false positives. Instead, dynamic analysis may be more suitable for this scenario.

### 4.3.4 Ambiguity of Natural Language

Coding conventions are normally described using the natural language. Although they are usually accompanied with examples of complying and non-complying code (e.g., see [46]), a certain degree of ambiguity still remains. Even when it is possible to provide a structured specification of the coding rule, it is challenging to determine whether such specification really represents what the original rule intended. This aspect is especially challenging for the evaluation of the metamodel, as discussed later in Chapter 7.

# Chapter 5

# CCSL Definition and Implementation

Working towards the realization of the workflow in Figure 4.1, In this section we describe the metamodel of the CCSL language, which is used to provide structured specifications of coding rules.

In this dissertation we decided to focus on Java since i) it is a very popular language, widely used in the industry, and ii) it is covered by many coding conventions.

## 5.1 Overview

In our approach, the structured specification of a rule specifies patterns that would *violate* the rule in the code. That is, given a rule $f\colon \mathcal{L} \to \{\texttt{valid}, \texttt{invalid}\}$, our objective is to give a specification of the subset of the programming language $\mathcal{L}_f \subseteq \mathcal{L}$ such that $f(\omega) = \{\texttt{invalid}\} \iff \omega \in \mathcal{L}_f$.

We identified the core concepts that need to be included in the language by analyzing multiple sources, including: i) existing coding conventions, in particular those for the Java language; ii) existing query languages; iii) main concepts of object-oriented programming; and iv) existing source code models, in particular the previously mentioned models Eclipse JDT DOM (an api to manipulate Java source code elements), the Modisco Java metamodel, and KDM.

Once the concepts have been identified, we defined the actual metamodel of our CCSL language using the Ecore metamodeling language, which is part of the Eclipse Modeling Framework (EMF) [48].

## 5.2 CCSL Metamodel

The main elements of the metamodel are described in the next sections, grouped by the *NamedElement*, *DataType*, *Expression*, and *Statement* packages. Its complete definition is available on GitHub [41].

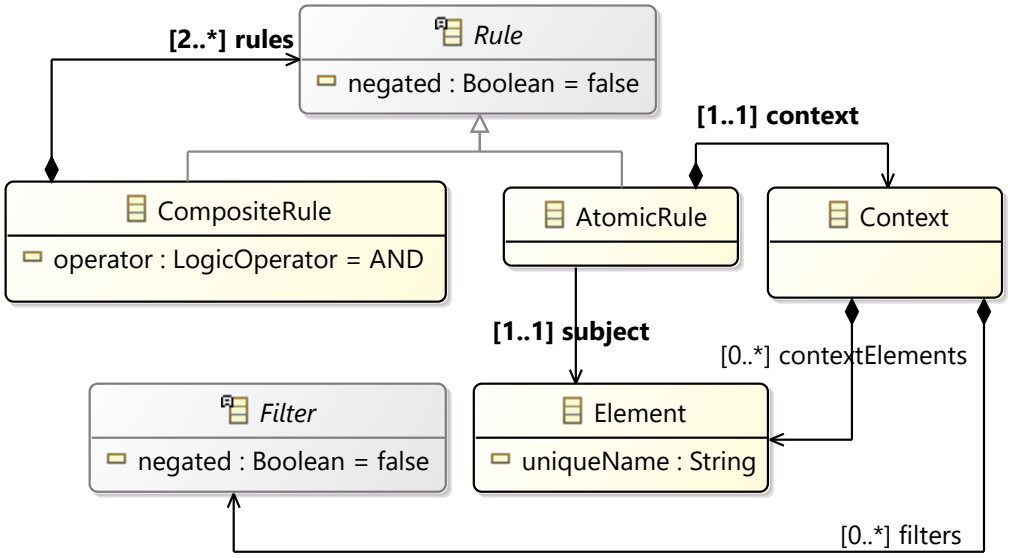



Figure 5.1: *Core* Package.

### 5.2.1 Core Package

This package contains the core concepts of the metamodel, which are illustrated in Figure 5.1 in EMF notation. In our metamodel, a coding rule is represented by the *Rule* metaclass, which can be either atomic or composite.

An *AtomicRule* is composed of:

**Context** The *Context* describes the pattern to be searched in the source code e.g., a class with name "Foo" that contains at least one method called as "qux". The context of a rule must contain at least one *Element* and it may contain a certain number of *Filter* instances.

**Subject** The *subject* of a rule identifies the programming language element to which the rule applies. The subject is always one of the elements defined in the *context*. In other words, the *subject* defines the element on which an alert is raised in case a violation is identified.

The *Element* metaclass is the top of the hierarchy of classes that represent different elements of the source code, e.g., classes, interfaces, methods, invocations, assignments, etc. The elements that constitute the context (and their relations) form the base for the pattern to be found in the code. The CCSL metamodel contains different subclasses of *Element*. Currently, the metamodel considers four kind of elements: *NamedElement*, *DataType*, *Expression*, and *Statement*. Metaclasses that extend *Element* are described in details in the next Sections.

Filters are used to retain only elements of the context that fulfill specific conditions e.g., to select only classes whose name is matched by a regular expression. *Filter* is an abstract metaclass, and it is extended by several concrete filters. A filter can be *negated*, which means that only elements *not* fulfilling the filter are selected. The full CCSL

metamodel contains different kind of filters; new ones can be added by creating a new subclass of *Filter*.

Complex rules can be specified as a *CompositeRule*, which is essentially a connector to combine multiple rules using Boolean logic operators ("and" and "or" operators).

### 5.2.2 NamedElements Package

A *NamedElement* is a *Element* that has a name given by the programmer (e.g., variables, classes, methods, interfaces, etc.). We have grouped *NamedElements* into the following categories:

**Custom Types**: This category includes the metaclasses that represent custom types (class, interfaces, annotations, and enumerations). Figure 5.2 illustrates the hierarchy of custom types.

The *TypeDeclaration* represents the top level of custom type hierarchy. The attribute "inheritance" defines whether a *TypeDeclaration* is "final" or "abstract" or "ANY" (it does not matter for the specification). A *TypeDeclaration* can be a *ComplexTypeDeclaration* (it represents custom types which can hold fields variables and methods as well as can implements/extends interfaces) or an *AnnotationType*. A *ComplexTypeDeclaration* can be a *JInterface* or a *ConstructComplexTypeDeclaration* (it represents complex types that can hold constructors). Finally a *ConstructComplexTypeDeclaration* is extended by the *JClass* and *JEnum* metaclasses.

**Variables**: This category includes all metaclasses which represent variables. Figure 5.3 illustrates the variable hierarchy.

A *Variable* represents the top level of the variable hierarchy and it is extended by the *InitializableVariable* (it represents any variable that can be initialized when it is declared) and *ParameterVariable* (it represents a variable as a parameter) metaclasses. Finally, the *InitializableVariable* is extendend by the *FieldVariable* and *LocalVariable*.

**Methods**: This category includes all metaclasses which represent methods. Figure 5.4 illustrates the method hierarchy.

A *SimpleMethod* represents the top level of the hierarchy and it abstracts commons concepts about the *Method* and *Constructor* metaclasses. Note that along the attribute "params" there is the attribute "paramsKind". The attribute "paramsKind" is used to define constraints about the "params" attribute. For example, if a method is specified as "Method(name = foo, params = {v1,v2})" then it will match to a method declared in source code such as "foo(v1,v2)", but also "foo(v1,v2,v3)". In case a method must has exactly a certain number of parameters, then the "paramsKind" value should be "EXACT".

### 5.2.3 DataType Package

A *DataType* is a *Element* that represents any kind of types in Java Language. There are two main kinds of *DataType*: *PrimitiveType* and *ObjectType* (it represents any other types that are not primitive types).

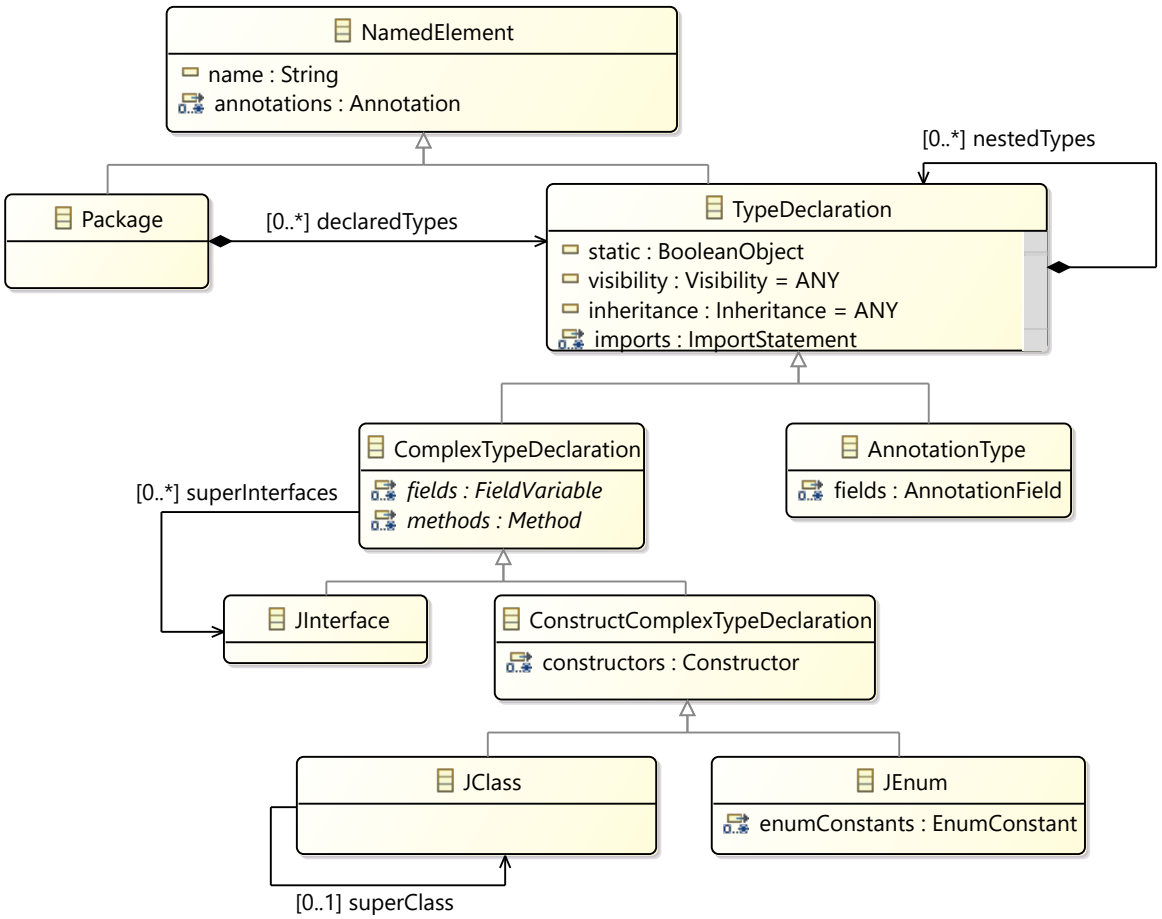



Figure 5.2: *CustomType* Hierarchy.

The *PrimitiveType* metaclass is extended by the known Java primitive types (int, short, long, double, byte, etc.). On the other hand, *ObjectType* is extended by the following metaclasses:

1. ArrayType. It represents an array of any *DataType*.
2. TypeDeclaration. It is the previously mentioned type that represents a custom type defined by the programmer (e.g., classes, interfaces, etc.).
3. *ParameterizedType*. It is an instance of a type that constains type arguments (e.g., *ArrayList<String>* is a *ParameterizedType* of the *ArrayList JClass*) .
4. *TypeParameter*. It is a generic type declared in methods (e.g., the *T* type of *public <T> voidfoo(T var)* is a *TypeParameter*)
5. *WildCardType*. It represents a wild card type of a type parameter (e.g., ArrayList<?> is a *ParameterizedType* of the *ArrayList JClass* using the *WildCardType*).

The *DataType* hierarchy is illustrated in Figure 5.5.

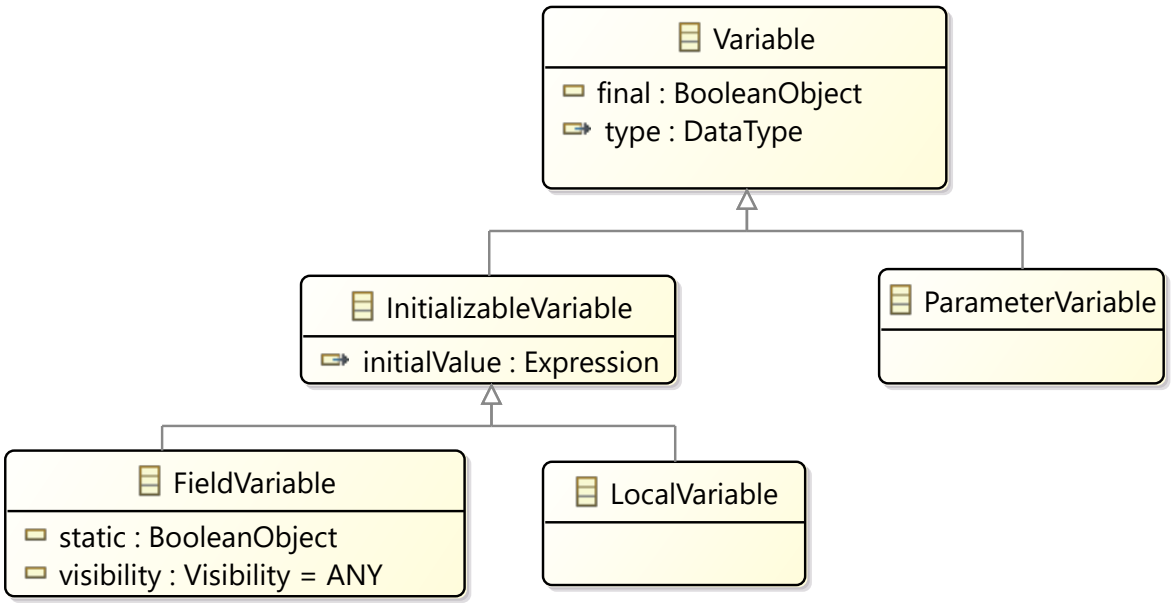



Figure 5.3: *Variable* Hierarchy.

### 5.2.4 Expression Package

The *Expression* metaclass represents elements that return values when they are evaluated (e.g., method invocations, assignments, cast expressions, strings concatenation, etc.). We decided to keep most of the Java expressions as they are, since it makes possible to create more specific rules. The current metamodel supports the following expressions:

1. VarDeclaration
2. CastExpression
3. TernaryExpression
4. InstanceofExpression
5. ArithmeticExpression
6. BooleanExpression
7. BinaryExpression
8. StringConcatenation
9. Invocation
10. MethodInvocation
11. ConstructorInvocation
12. AnonymousClassCreation
13. VariableAccess
14. DataTypeAccess
15. ArrayCreation
16. ArrayIndexesAccess
17. ArrayInitializer
18. SimpleAssignment
19. Assignment
20. UnaryAssignment
21. LiteralValue
22. NullLiteral
23. CharacterLiteral
24. StringLiteral
25. NumberLiteral
26. BooleanLiteral

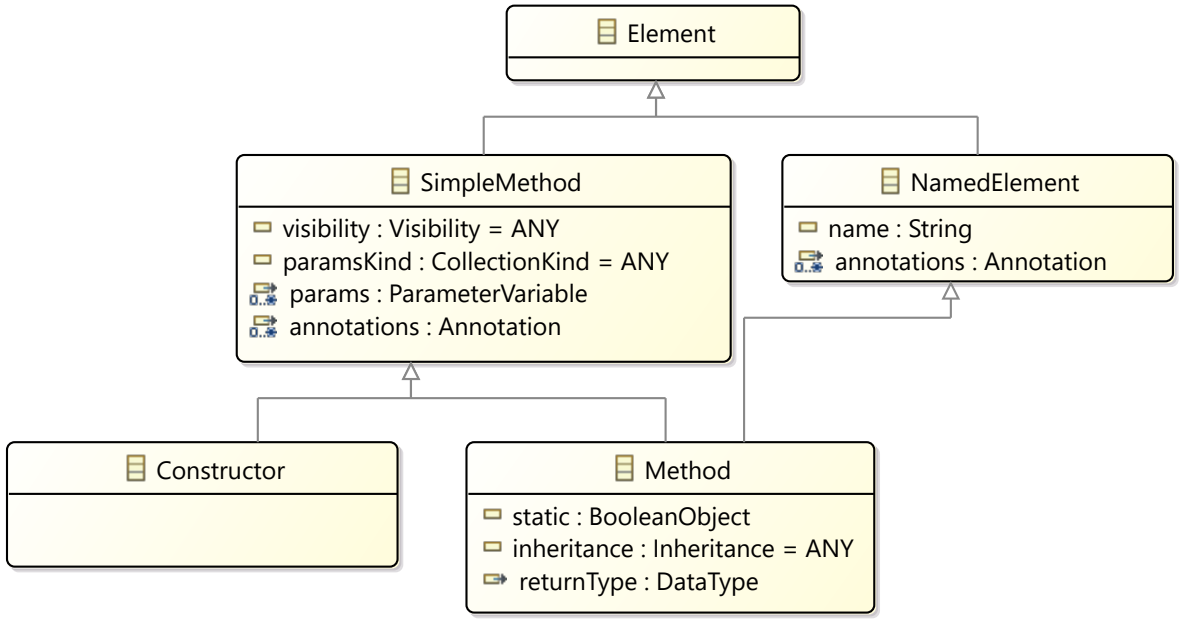



Figure 5.4: *Method* Hierarchy.

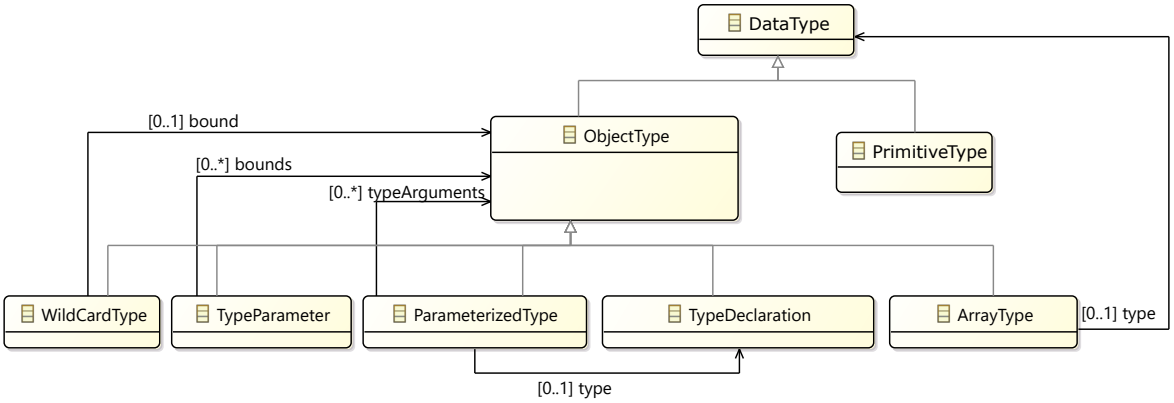



Figure 5.5: *DataTypes* Package.

Most of the listed expressions are self-explanatory (e.g., *CastExpression* represents a cast expression). However, few of them may not be understandable by only reading their names:

1. Invocation is a generic metaclass that is extended by MethodInvocation and ConstructorInvocation. Invocation is useful when it is necessary to express an invocation independently of whether it is a method invocation or a constructor invocation.

2. VariableAccess represents the access to the reference of a variable. For example, the declaration *int* $b = c$ access the $c$ variable in order to get its value.

3. DataTypeAccess represents the access of a class by its name. For example, the invocation *ClassA.foo()* access the *ClassA*.

4. ArrayInitializer is an alternative way to initialize an array. For example, the right hand of the assignment *String[] array* = *{"a", "b", "c"}* is an array initializer.

5. SimpleAssignment is a generic metaclass that is extended by Assignment and UnaryAssignment. In our metamodel, we consider both *a = a + 1* and *a++* being an assignment. The first one is represented by the *Assignment* metaclass while the second is represented by the *UnaryAssignment*.

### 5.2.5 Statements Package

The *Statement* metaclass are the commands to be executed when the code is being executed (e.g., if, while, for, try-catch, etc.). In order to give more flexibility to the modeler to create CCSL, we have specified almost all Java statements:

1. ControlFlowStatement
2. ConditionalControlFlowStatement
3. LoopStatement
4. IfStatement
5. ForStatement
6. ForEachStatement
7. WhileStatement
8. DoWhileStatement
9. SwitchStatement
10. Block
11. SynchronizedBlock
12. ThrowStatement
13. EmptyStatement
14. ReturnStatement
15. BreakStatement
16. ContinueStatement
17. AssertStatement
18. TryStatement
19. CatchClause
20. ImportStatement

### 5.2.6 Filters Package

The *Filter* metaclass represents filters that are used to identify specific elements within the scope given by the *element* attribute of the *Rule* (see Figure 5.1).

The structure of a filter is depicted in Figure 5.6. To improve flexibility, filters adopt the main idea of the Composite design pattern [16], where the *CompositeFilter* represents a list of filters combined by a logic operator (and, or operators), and *AtomicFilter* is an abstract metaclass which represents an entry-point to define new filters. Every filter can also be *negated* or not.

Every *AtomicFilter* contains a list of elements to which the filter will be applied (*target* attribute). Concrete filters for different purposes are created by extending the *AtomicFilter* abstract metaclass (see Figure 5.6). The current metamodel supports the following filters:

1. CountFilter. It counts how many times a sample *s* appears in a target object *obj*.
2. BlockFirstStatementFiter. It checks if an object *f* appears as first element in a target *Block* instance *b*.

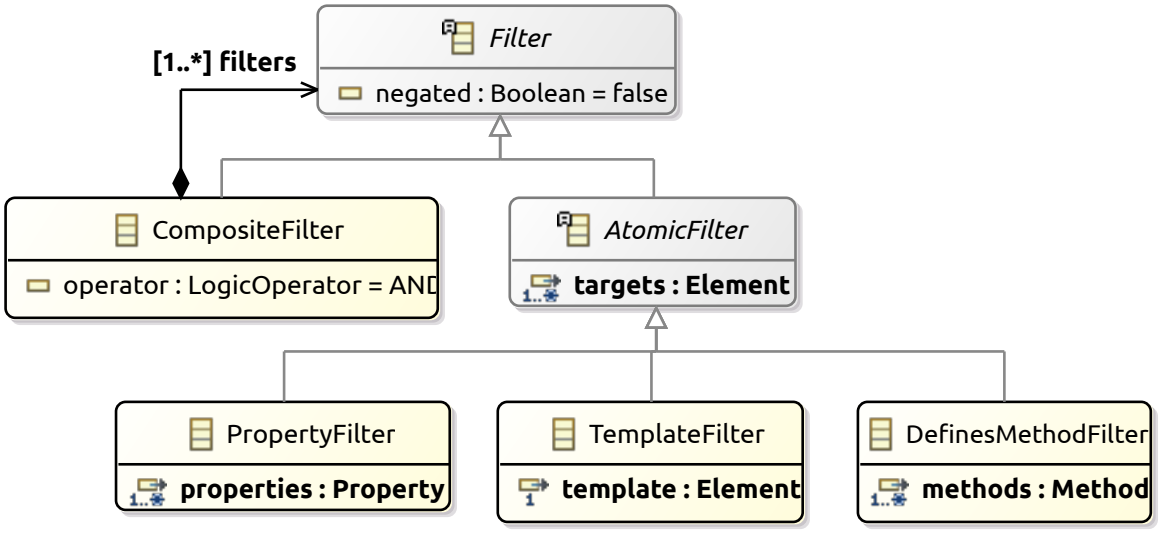



Figure 5.6: *Filters* Package

3. HasSameReferenceFilter. It checks if its targets are the same element in the source code.

4. HasSubclassFilter. It checks recursively if a class *c* has a *sub* subclass.

5. HasSuperclass. It checks recursively if a class *c* has a *super* superclass.

6. ImplicitContainerFilter. It checks recursively if an *obj* object has an implicit container *c*.

7. ImplicitContentFilter. It checks recursively if an *obj* object has an implicit content *c*.

8. IsKindOfFilter. It checks if a *DataType* or an *Expression* can be considered as type *t*. For example, an object of type Foo, which extends a Qux class can be considered as Qux.

9. IsTypeOfFilter. It checks if a *DataType* or an *Expression* is of type *t*. For example, an object of type Foo, which extends a Qux class is of type Foo, but not Qux.

10. BlockLastStatementFilter. It checks if an object *f* appears as last element in a target *Block* instance *b*.

11. RegexFilter. It checks if the name of an *NamedElement* element matches a specific regular expression.

12. SameNameFilter. It check if its target have the same name.

13. *TemplateFilter*. This filters checks if the target element can be matched to the provided template, which is another instance of the *Element* metaclass. This is the most generic filter and it was designed to improve the flexibility of the metamodel. In fact, in case none of the available filters can be used to explicitly specify the desired condition, the *TemplateFilter* can be used to provide a "sample" instance of an *Element* that describes the kind of elements that should be selected.

Examples of application of filters are presented later in Section 5.3.

## 5.3 Writing a CCSL Specification

To better explain how CCSL is actually used, in the following we provide concrete examples of application of the metamodel. We separated this sections into three subsections, each one containing describing the main features of CCSL. A broader study has been executed and it is reported in Chapter 7.

### 5.3.1 Getting started with CCSL specification

As previously mentioned, the main part of a rule is its subject, which identifies the kind of elements to which it applies. It should be noted that, while an *Element* may in general have many different attributes, most of them have a minimum multiplicity of zero. This means that only the attributes that are needed for expressing the rule need to be specified, and all the other may be ignored.

Consider the rule *AvoidInstanceOfChecksInCatchClause* from PMD rules:

"*AvoidInstanceOfChecksInCatchClause: Each caught exception type should be handled in its own catch clause.*"

We have specified the above rule by considering a code portion invalid when it contains an *instanceof* statement checking the type of a *variable* declared as a parameter in the *catch* clause. The Figure 5.7 illustrates a Java code which violates such a rule.

```
1 try {
2   /* code */
3 } catch (Exception ee) {
4   /* code */
5   if(ee instanceof IOException) { //violation
6     /* code */
7   }
8   /* code */
9 }
```

Figure 5.7: A piece of code illustrating a violation of the *AvoidInstanceOfChecksInCatchClause* rule

Note that there are other scenarios that could be consider to violate the same principle behind the rule. For example *ee.getClass() == IOException.class* is also a clearly violation of the rule. However, we only specified as violations the above illustrated case since our goal is to try to match exactly our specification to the description of the rule in the PMD manual, since we planned to make an experiment comparing our specifications results to the PMD results (see Section 7).

The CCSL specification of the *AvoidInstanceOfChecksInCatchClause* rule is illustrated in Figure 5.8. We specified the rule using the *AtomicRule* metaclass. The rule *subject* is an *InstanceofStatement* where its *objectExpression* (the left side of a instanceof statement) is a reference to a variable (*VariableAccess*) that points to the *ParameterVariable* declared in the *CatchClause*.

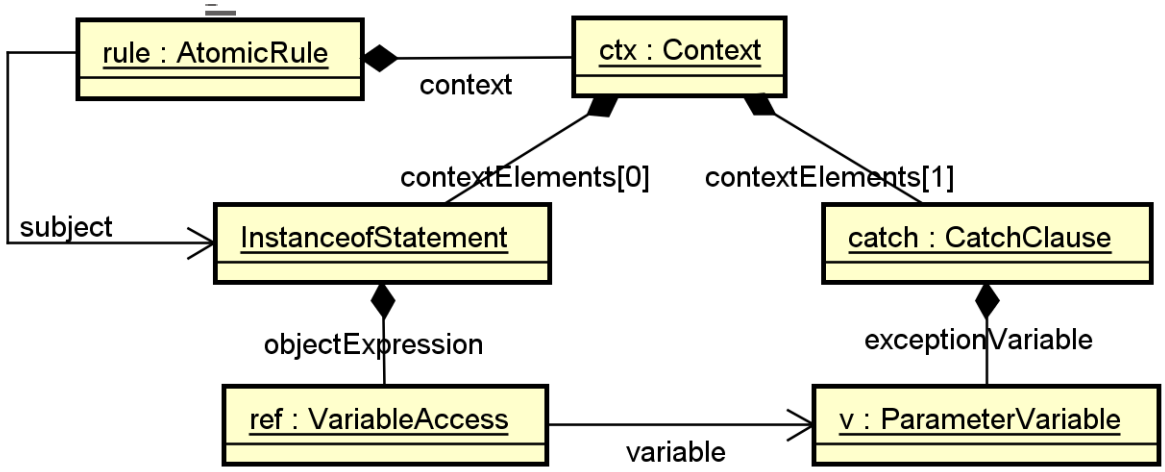



Figure 5.8: *AvoidInstanceOfChecksInCatchClause* CCSL specification

Another example to introduct to CCSL specification can be the THI00-J. rule of the SEI CERT Coding Standard for Java [46]:

"*THI00-J. Do not invoke Thread.run(): Thread startup can be misleading because the code can appear to be performing its function correctly when it is actually being executed by the wrong thread. Invoking the Thread.start() method instructs the Java runtime to start executing the thread's run() method using the started thread. Invoking a Thread object's run() method directly is incorrect. [. . . ]*"

The above rule can be specified using CCSL as illustrated in Figure 5.9. In this specification a code portion is considered invalid if it contains an invocation of the *run* method of the *Thread* class. We specified the rule using the *AtomicRule* metaclass. The rule *context* is defined by creating an instance of the *MethodInvocation* metaclass taking the reference of the *run Method* of the *Thread JClass*. The *subject* is the *MethodInvocation*, which means that every method invocation of the thread run method in the source code will be marked as a violation.

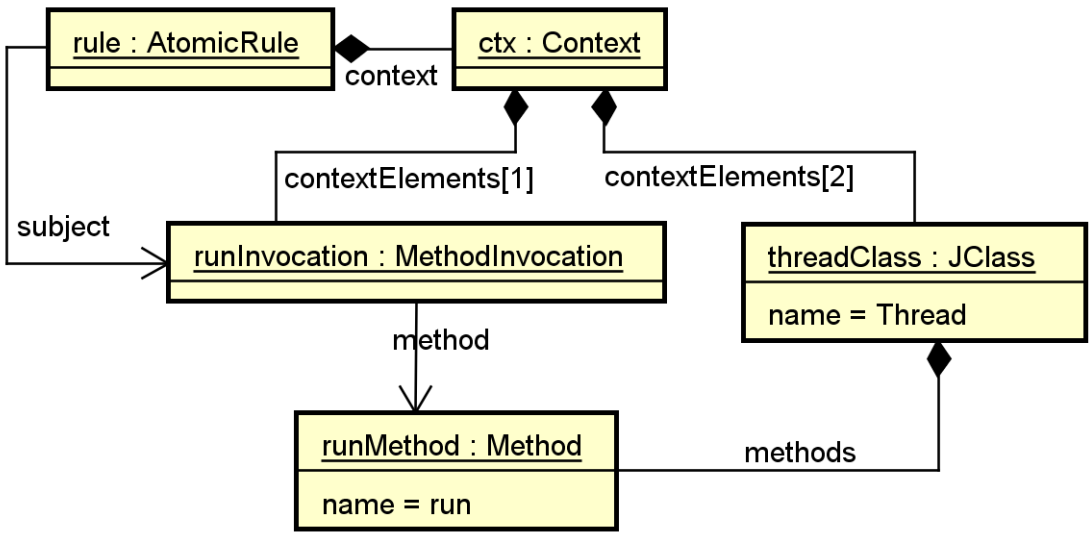



Figure 5.9: *THI00-J. Do not invoke Thread.run()* CCSL specification

```
1 public class SampleClass {
2   public static void main(String[] args) {
3     new java.lang.Thread(/* args */).run();
4     new my.another.Thread(/* args */).run();
5   }
6 }
```

Figure 5.10: A piece of code illustrating a false positive of the previously CCSL specification

While the above specification can be considered correct, we still can improve the specification. Imagine an unusual scenario where a programmer creates a class named as *Thread* and he also defines a method named as *run* inside it. Therefore, every invocation of the *run* method of the "another" *Thread* will also be marked as a violation since it matches with the given specification. Figure 5.10 illustrates such a scenario where both lines three and four are reported as violations of the rule.

To avoid such false positives, we can also refine the specification, i.e., providing more details to the specification. There are at least two kinds of details that we can provide in this situation:

1. Providing the full name of the *Thread* class, i.e., *java.util.Thread*.
2. Providing more details about the *run* method, i.e., it is a non-static public method with no arguments that does not return anything (void).

The Figure 5.11 illustrates the refined specification. Now a code portion is considered invalid if it contains an invocation of the *run* method (a non-static public method that does not have any arguments and does not return anything) of a *java.util.Thread* class. Note that in the new specification we added the attribute *paramsKind (EXACT)* in the *run* method, this means that only implementations of the *run* method that contains exactly zero parameters will be considered.

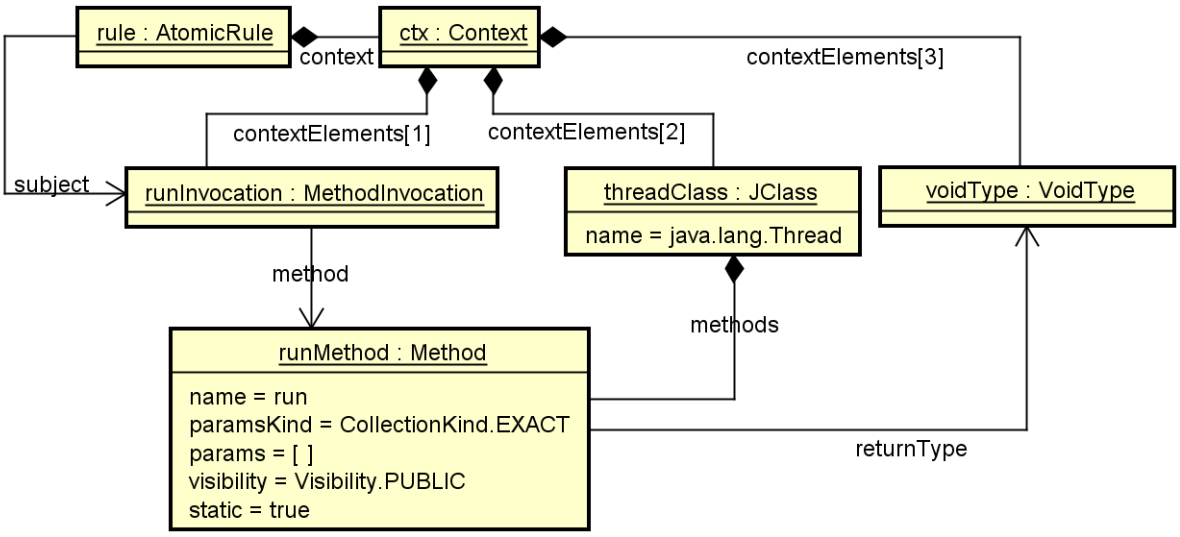



Figure 5.11: *THI00-J. Do not invoke Thread.run()* CCSL refined specification

### 5.3.2 Using Filters

Filters are used to identify specific elements according to a criteria. As well as rules, some *Filters* can have their own context. The context of a filter includes information that is related to the filter condition but it is not actually part of the rule context. For example, consider the rule MET09-J of the SEI CERT Coding Standard for Java [46]:

"**MET09-J:** ***Classes that define an equals() method must also define a hashCode() method.*** *[...] The equals() method is used to determine logical equivalence between object instances. Consequently, the hashCode() method must return the same value for all equivalent objects. Failure to follow this contract is a common source of defects."*

Note that, in general, determining if the *hashCode* method actually returns the same value for all equivalent objects is not feasible with static analysis, and it is in general a hard problem. Therefore, we do not include it in the specification of the rule.

Figure 5.12 illustrates the specification of the above rule using CCSL. The element to be searched, which defines the *subject* of the rule, is a *JClass* that contains a method named "equals". However, only classes that define an "equals" method and do not define a "hashCode" method must be matched. This can be achieved by applying a filter: the *TemplateFilter* receives as target the *JClass* and checks whether it is *not* possible (it is negated) to match it against the template (a *JClass* that contains the "hashCode"). Note that the *template* is declared in the *TemplateFilter* context and additional informations related only to the *TemplateFilter* should be added in its context.

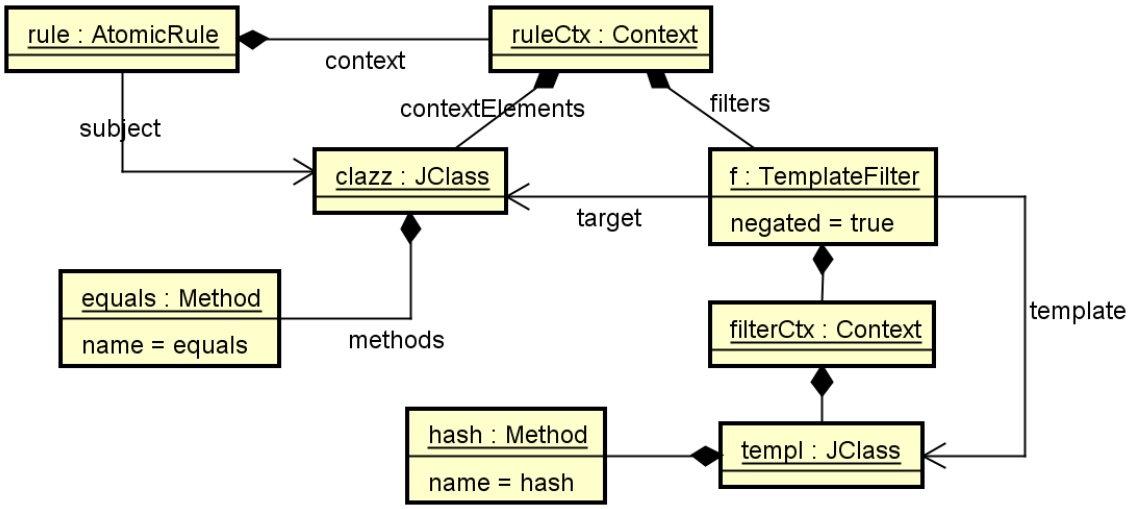



Figure 5.12: MET09-J CCSL specification

Rule MET09-J is also a good example of how rules defined in natural language may be ambiguous and thus be interpreted in different ways. There are at least two aspects of this rule that are ambiguous:

1. It is not clear which signatures of the *equals* method and *hashCode* method that are affected by the rule. In fact, it is possible to define multiple methods having the same name, using polymorphism. The traditional signature of the equals method in Java is *boolean equals(Object obj)*. However, it is possible to overload the equals method by defining a slight variation: *boolean equals(CustomClass obj)*. Whether

the rule MET09-J should apply only to the original *equals* method or not is open to interpretation.

2. The *hashCode* method could have been defined in a superclass rather than in the same class that defines the *equals* method. This would also prevent introducing the bug mentioned by the rule. However, by looking at the text it is not clear whether such implementation should raise a warning or not.

Figures 5.13 illustrates an alternative specification of the MET09-J rule, which considers the following interpretation. If a certain class defines a *boolean equals(Object obj)* method, then the *int hashCode()* method should be provided in that class or in one of its superclasses, except for the *Object* class.

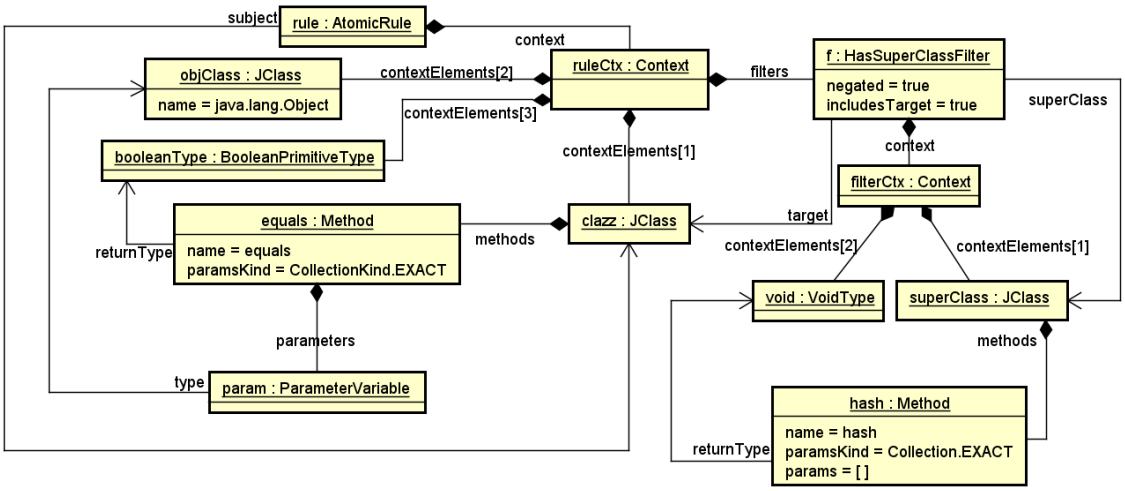



Figure 5.13: MET09-J refined CCSL specification

The *subject* of the rule is still a *JavaClass*, but now the *equals* method is specified as a method which returns a *boolean* and contains exactly one parameter of type Object. Finally, the filter being applied in the scope is the *HasSuperClassFilter*, which recursively checks whether its target does *not* have (it is negated) a super class that defines the method *int hashCode()*. Because the attribute *includesTarget* of the *HasSuperClassFilter* is setted as true, the check is also performed on the *target* of the filter itself, that is, it is also checked if the current class does not contain a method named "hashCode".

In this case, whether specification (a) or (b) is the correct one is debatable. This however highlights the importance of providing a structured semi-formal specification, to avoid ambiguities of this kind.

### 5.3.3 CCSL Relations

An important feature of the CCSL is the possibility to specify relations between elements. When an object is referenced or created inside of another object, this means that the derived checkers will look for an immediate relation between these objects. To better understand this concept, consider the PMD rule *AvoidInstantiatingObjectsInLoops*:

"***AvoidInstantiatingObjectsInLoops.*** *New objects created within loops should be checked to see if they can created outside them and reused.*"

In the above rule, an alert is reported every time that an instantiation is detected in a loop. The Figure 5.14 illustrates three examples of violations of such a rule.

```
while(/* condition */)
  new Object();
```

(a)

```
while(/* condition */) {
  new Object();
}
```

(b)

```
while(/* condition */) {
  if(/* condition */) {
    new Object();
  }
}
```

(c)

Figure 5.14: Violations of the *AvoidInstantiatingObjectsInLoops* rule

A incomplete CCSL specification to the *AvoidInstantiatingObjectsInLoops* rule is illustrated in Figure 5.15, where the relation "LoopStatement{body = ConstructorInvocation}" is established. Contrary to what one might think, this specification only reports Figure 5.14a as a violation. This occurs because in the structured model corresponding to Figures 5.14b, 5.14c the loop body is a *Block* metaclass. In other words, when we create the relation "LoopStatement{body = ConstructorInvocation}" this mean that the LoopStatement body must be a ConstructorInvocation and not that a ConstructorInvocation must be in LoopStatement body somewhere.

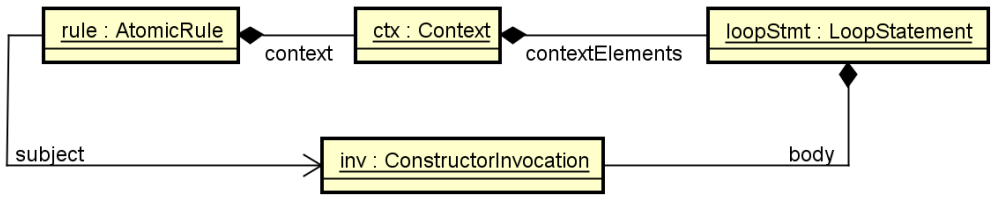



Figure 5.15: Mistaken CCSL specification of *AvoidInstantiatingObjectsInLoops* rule

When we want to specify that an element can be inside another at any level of depth we can use the *ImplicitContainerFilter*. Such a filter recursively gets all the containers of its target trying to match one of them to a sample element passed to the filter. The Figure 5.16 illustrates the correct specification using the *ImplicityContainerFilter* of the *AvoidInstantiatingObjectsInLoops* rule.

According to the new specification illustrated in Figure 5.16, now a code portion is considered invalid if it contains a *ConstructorInvocation* (an instantiation) where such a constructor is implicitly a content of any *LoopStatement*. In other words, the correct specification will raise an alert on all the three codes of Figure 5.14.

### 5.3.4 Support to Naming Conventions

CCSL also supports the specification of some naming conventions by using filters. For example, consider the rule TestClassWithoutTestCases of the PMD Error prone set rules:

"**TestClassWithoutTestCases:** *Test classes end with the suffix Test. Having a non-test class with that name is not a good practice, since most people will assume it is a test case.*"

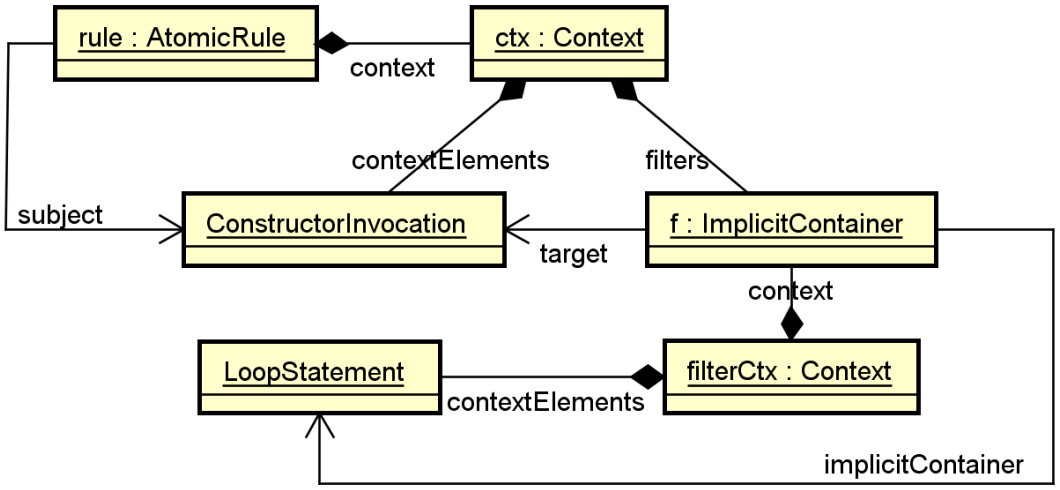



Figure 5.16: Alternative specification of *AvoidInstantiatingObjectsInLoops* rule

We have specified this rule by invalidating a class that ends with the suffix Test that does not have any method annotated with the Test annotation. Figure 5.17 illustrates the specification of the above rule. The *RegexMatchFilter* defines the regular expression *(.)*Test*, which checks if a name of a class has the suffix "Test". The *TemplateFilter* checks if the target class does not have any method annotated with the "Test" annotation. When two or more filters are specified without using the connector (*ComposedFilter*), then all filters are connected by the "and" operator by default.

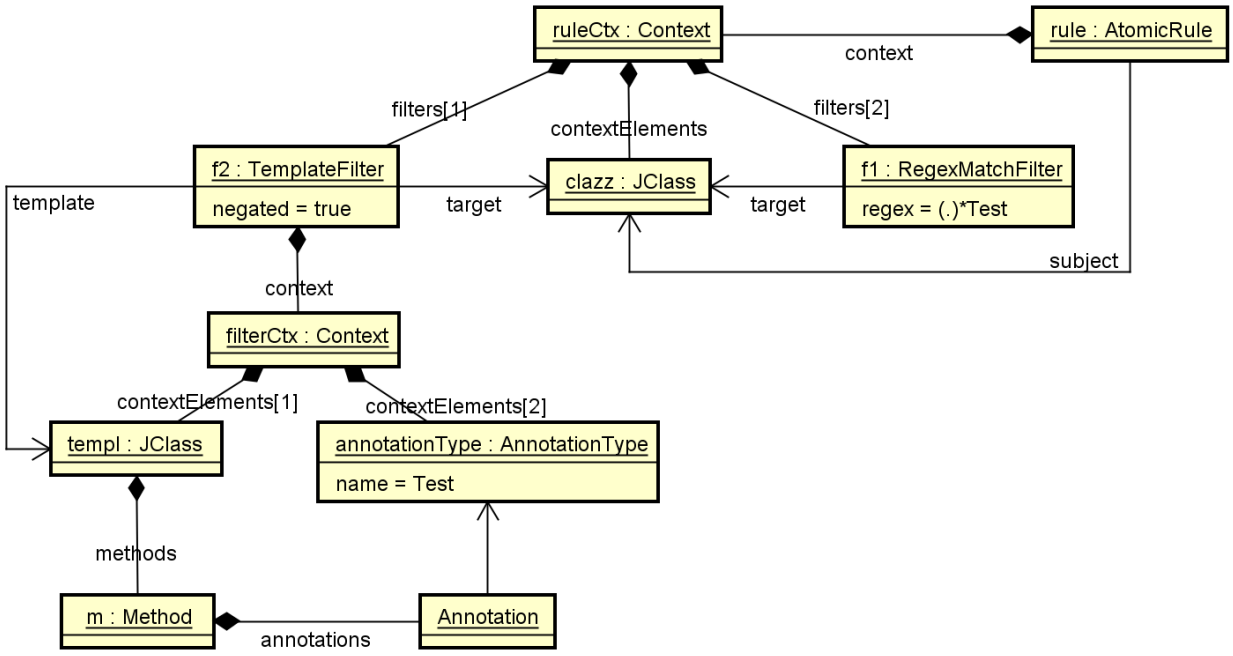



Figure 5.17: TestClassWithoutTestCases CCSL specification

As this Chapter presented the CCSL metamodel, in the next Chapter we show how we defined the model-to-text transformation that generate OCL queries from the CCSL specification.

# Chapter 6

# CCSL Model-to-Text Transformation

This section is separated into two subsections. The first describes the transformation algorithm (model-to-text transformation) to concretely generate OCL queries from CCSL specifications. The second presents an example to better understand the generation flow.

## 6.1 Transformation Algorithm

The model-to-text transformation has been developed using the Acceleo language [1]. The Acceleo language is not a object oriented language. Instead, it is composed by modules where each module exposes templates and queries.

The structure of the transformation is illustrated in Figure 6.1. The main module, i.e., the entry-point to the generation of an OCL query from a CCSL specification is the *OclBuilder* module. The generation flow of an OCL query can be summarized in three steps:

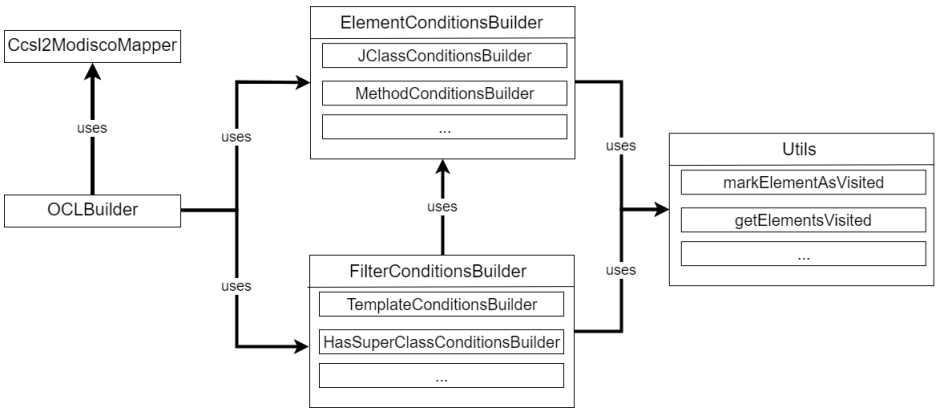



Figure 6.1: Acceleo Modules to transform CCSL specification to OCL query

1. *Mapping CCSL metaclass to MoDisco Java metaclass.* We recall that a rule *subject* defines the element on which an alert is raised in case a violation is identified. We first access the *Ccsl2ModiscoMapper* module in order to get which metaclasses from the structured Java model can represent the rule *subject* metaclass from CCSL specification. With this information, we then collect all instances of the mapped metaclasses in the Java model and proceed to the next step.

2. *Subject Conditions.* In this stage we filter the selected instances of the previous step by creating conditions to the rule *subject* properties and relations defined in the CCSL specification. Here the module *ElementConditionsBuilder* works as a Façade where it creates all conditions of a given *Element* to satify the properties and relations defined in the specification.

   The algorithm is based on a graph depth-first search where we start in the *subject* element. We then recursively write OCL conditions while we are navigating to the declared attributes and relations in the CCSL specifications. Whenever a element is visited, the following actions are done:

   (a) We mark the element as visited to avoid infinite loops, i.e., we do not generate element conditions if it is already visited.

   (b) We declare a variable name for the element visited in the OCL query. This is useful in situations where we need to visit a element that has been visited already. In this case, we just reference the element by its variable name.

   (c) We generate conditions for the declared attributes in the element being visited as well as we navigate through the element relations by recursively visiting another elements. We also visit the element container in case it has not been visited yet.

   The actions to get all the elements visited as well as to generate an unique name to a visited element are provided by the *Utils* module.

3. *Filters conditions.* In this stage we access the module *FiltersConditionsBuilder* that works as a Façade to generate OCL constraints of a given *Filter*. Each filter has its own strategy for generating the OCL, since each one has a specific purpose.

## 6.2 Generated OCL Example

Consider the previously mentioned specification of the rule "AvoidInstanceOfCheckInCatchClause" illustrated in Figure 5.8. The Figure 6.2 illustrates the OCL generated following the steps:

1. *Mapping CCSL metaclass to MoDisco Java metaclass.* The rule subject is a CCSL *InstanceofExpression* metaclass and it is directly mapped to the MoDisco *InstanceofExpression* metaclass (they have the same name). We then create a set containing all the instances of MoDisco *InstanceofExpression* metaclass (line 1) and we proceed to the next step.

2. *Subject Conditions (visiting InstanceofExpression).* Here we create OCL conditions to satisfy the declared attributes and relations of the rule *subject*. We first generate an unique name to reference the *InstanceofExpression* instance (instanceOfExp_1, line 1). We then navigate through the relation *objectExpression* declared in the CCSL specification in order to visit the *VariableAccess*.

3. *Subject Conditions* (visiting *VariableAccess*). This instance is referenced as *varAccess_1* in the OCL query (line 2). Note that the generated checkers skip by default parenthesis expressions, so it does not matter if the instanceof expression is formatted such as "ee instanceof Type" or "(ee) instanceof Type" (lines 2–9). A CCSL *VariableAccess* is mapped to the *SingleVariableAccess*, *FieldAccess*, and *SuperFieldAccess* MoDisco metaclasses. Because the *VariableAccess* is referencing a *ParameterVariable* in the CCSL specification, we know that the *VariableAccess* should only be matched to the *SingleVariableAccess*, therefore it is created an OCL condition to check the *varAccess* type (line 10). We now visit The *ParameterVariable* by navigating through the *variable* relation established between the *VariableAccess* and the *ParameterVariable* instances in the CCSL specification.

4. *Subject Conditions* (visiting *ParameterVariable*). The *ParameterVariable* is mapped to the *SingleVariableDeclaration* MoDisco metaclass. This instance is referenced as *paramVar_1* (line 11). We then create a type check condition in order to check the type of *paramVar_1* (line 13). We proceed to one more step by visiting the *ParameterVariable* container (*CatchClause*) using the *oclContainer* function (line 14).

5. *Subject Conditions* (visiting *CatchClause*). This is the last step as all another specified CCSL elements have been visited already. The CCSL *CatchClause* metaclass is directly mapped to the MoDisco *CatchClause* metaclass. We generate an unique name to reference the *CatchClause* (*catchClause_1*, line 14) and it is also generated a type check condition since the *oclContainer* returns an *OclAny* type (line 15). We finish the generation by creating a last condition to ensure that *catchClause_1* and *paramVar_1* is connected through the *exceptionVariable* relation (line 16).

```
1  InstanceofExpression.allInstances()→select(instanceofExp_1: InstanceofExpression|
2    let varAccess_1: ASTNode = instanceofExp_1.leftOperand→asOrderedSet()→closure(
3      v: ASTNode |
4      if v.oclIsKindOf(ParenthesizedExpression) then
5        v.oclAsType(ParenthesizedExpression).expression
6      else
7        v
8      endif
9    )→last() in varAccess_1 <> null and
10   varAccess_1.oclIsKindOf(SingleVariableAccess) and
11   let paramVar_1: ASTNode = varAccess_1.oclAsType(SingleVariableAccess).variable in
12   paramVar_1 <> null and
13   paramVar_1.oclIsKindOf(SingleVariableDeclaration) and
14   let catchClause_1: OclAny = paramVar_1.oclContainer() in catchClause_1 <> null and
15   catchClause_1.oclIsKindOf(CatchClause) and
16   catchClause_1.oclAsType(CatchClause).exception = paramVar_1
17 )
```

Figure 6.2: The generated OCL from AvoidInstanceofInCatchClause specification

## 6.3 GUI CCSL Checker Prototype

We have implemented a prototype to execute the CCSL checkers in Java projects. The implementation is an Eclipse-plugin and it accepts both CCSL specifications as well as

the generated OCL files (it is not necessary to generate an OCL file again if it is available already) as rules input.

The Figure 6.3 illustrates the main window of the prototype where the user should select: i) the Java projects to run the rules; ii) the rules to be executed (both OCL generated files and CCSL specifications can be selected); and iii) the output folder in which the report output will be sent.

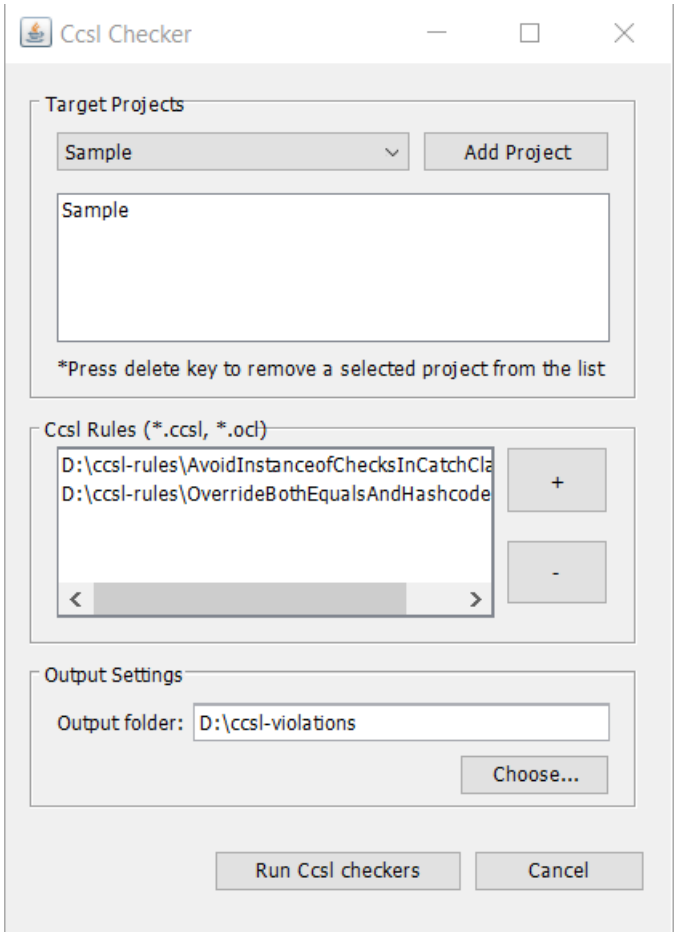



Figure 6.3: Main window of CCSL checker.

When the user presses the "Run CCSL Checker" button, the window illustrated in Figure 6.4 appears to indicate the progress execution. The prototype executes the following steps: i) it extracts the java model of all selected projects; ii) it generates the OCL queries of selected rules; iii) it executes the OCL queries in each selected project; and iv) it generates a *txt* file per rule containing the violations found in the select projects (each violation contains the Java file path and a line where the violation has occurred).

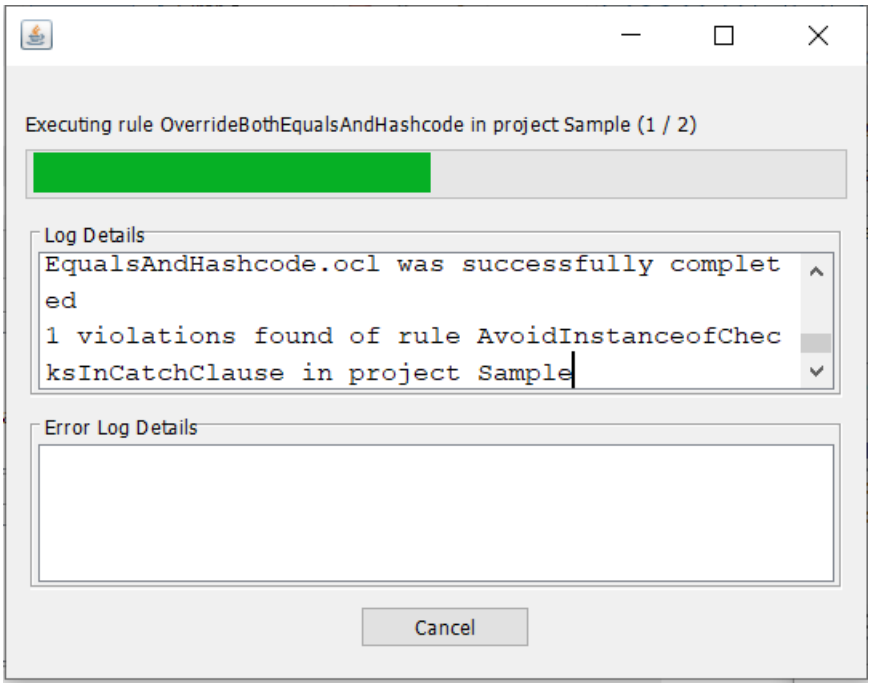



Figure 6.4: Window indicating the progress of the execution of the rules in the selected projects

# Chapter 7

# Approach Evaluation

In this section we report on the experiment we performed as an evaluation of the proposed approach. The experiment was divided into two experiments, each designed to address the following research questions:

***RQ #1 - Does CCSL can be use to specify existing coding rules?*** This question seeks to understand whether the current CCSL metamodel can be used to specify existing coding rules from real static analysers.

***RQ #2 - Is it possible to derive automated checkers from the CCSL Specification?*** This question seeks to understand whether the current CCSL specifications can be transformed to checkers that verify the source code as well as whether the generated checkers provide the same result of the existing static analysis tool.

## 7.1 CCSL Metamodel Evaluation

### 7.1.1 Methodology

We selected rules from two popular coding conventions for the Java language. Each rule in the selected subsets was manually analyzed, and a specification using CCSL was attempted. Whether a correct specification can be provided or not is somehow subjective, as it depends on the skill of the modeler, and on the interpretation of the original rule written in the natural language. To decide if a rule was correctly specified we used the following two heuristics:

- *Have all the concepts mentioned in the original rule been included in the CCSL specification?*
- *Does the CCSL specification contain enough information to verify the rule automatically?*

Otherwise, we consider that we were not able to provide a specification of the rule using our language.

### 7.1.2 Selected Coding Rules

For this evaluation we selected two popular but quite different set of rules: one is derived from checks implemented in a static analysis tool, while the other contains general principles for improving security, for which a checker is not necessarily available.

**PMD Rules.** The PMD tool [4] mentioned before is a static code analyzer that provides a wide set of predefined coding rules, organized according to different topics. We selected three groups of rules for our evaluation: *Error Prone*, *Multithreading*, and *Performance*. All of these rules are described with natural language, and then implemented in the tool by means of an XPath query or by explicitly coding a Java method that performs the check. Due to the characteristics of the tool, it is not possible to implement rules spanning the whole source tree, or rules requiring complex analysis. Therefore, coding rules contained in this set are in general relatively simple.

**SEI CERT Coding Conventions.** The Software Engineering Institute (SEI) of Carnegie Mellon University maintains extensive coding conventions focused on security, for popular languages such as Java and C++ [46]. Rules in this collection are harder to model, as they address different aspects of programming, and their descriptions span different levels of detail. For example, some rules go into the details of the unpacking of compressed files (IDS04-J)[1], while other ones simply give the generic recommendation to "perform proper cleanup at program termination" (FIO14-J)[2]. We excluded some subsets from our evaluations, as they are clearly outside the scope of the metamodel, for example the *Runtime Environment* subset, which includes recommendations for the configuration of the runtime platform (e.g., ENV04-J: "Do not disable bytecode verification"). The selected subsets are listed in Table 7.3.

### 7.1.3 Results

Results of the analysis are reported in Table 7.3. The CCSL specifications given for the analyzed rules are also available on GitHub [41].

During the experiment we analyzed a total of 216 individual coding rules. That is, we tried to devise the CCSL specification of 216 different rules, similarly to what was done for the MET09-J rule in Section 5.3.2.

The results confirm the feasibility of the approach, as we were able to specify more than half (63%) of the considered rules. However, they also highlight limitations of the current version of the language, and they indicate that the kind of rules that is considered has a great impact on the chance of a successful specification.

In fact, we were able to specify most of the rules provided by PMD (80%) using our approach. This was somehow expected, as such rules have already an implementation that

[1] `https://wiki.sei.cmu.edu/confluence/display/java/IDS04-J.+Safely+extract+files+from+ZipInputStream` (Accessed October 24, 2019)

[2] `https://wiki.sei.cmu.edu/confluence/display/java/FIO14-J.+Perform+proper+cleanup+at+program+termination`

Table 7.1: Number of individual rules that were successfully specified using CCSL (*Specified/Yes* column), and those for which a specification was not possible (*Specified/No* column).

| **Source** | **Section** | **Specified** | | **Total** |
|---|---|---|---|---|
| | | *Yes* | *No* | |
| PMD | Error Prone | 79 (81%) | 19 (19%) | 98 |
| | Multithreading | 8 (80%) | 2 (20%) | 10 |
| | Performance | 24 (80%) | 6 (20%) | 30 |
| | **SubTotal** | **111 (80%)** | **27 (20%)** | **138** |
| SEI CERT | Characters and Strings | 0 | 5 (100%) | 5 |
| | Declarations and Initialization | 0 | 3 (100%) | 3 |
| | Exceptional Behavior | 4 (40%) | 6 (60%) | 10 |
| | Expressions | 1 (17%) | 5 (83%) | 6 |
| | Methods | 7 (54%) | 6 (46%) | 13 |
| | Numeric Types and Operations | 3 (25%) | 9 (75%) | 12 |
| | Object Orientation | 5 (42%) | 7 (58%) | 12 |
| | Thread APIs | 5 (83%) | 1 (17%) | 6 |
| | Thread Pools | 0 | 5 (100%) | 5 |
| | Visibility and Atomicity | 0 | 6 (100%) | 6 |
| | **SubTotal** | **25 (32%)** | **53 (68%)** | **78** |
| **Total** | | **136 (63%)** | **80 (37%)** | **216** |

is capable of checking them, meaning, at least, that their verification can be automated. It should be noted however that with PMD a new checker must be manually written for each new rule and that some checkers are just Java code, i.e., they do not have a high-level specification. Rules provided by PMD for which it was not possible yet to provide a specification (20%) are due to the lack of specific primitives in the current version of the metamodel. For example, the analysis highlighted that more complex concepts related to the execution flow are needed.

Conversely, we can provide a CCSL specification only for a small portion of the rules in the SEI CERT group. Most of those rules are in fact either too generic or so specific that it would be necessary to develop specific checkers to verify them. Increasing the abstraction level of the metamodel may improve these results, although this may make the generation of artifacts from the metamodel harder. At the same time, we highlight that for many of the SEI CERT rules no automated checker is available anyways. Considering this aspect, being able to provide a structured, machine-readable, specification of almost one third of them as a first attempt is still an encouraging result.

Finally, it should be noted that those results represent a study based on the current status of our work. Further research can improve the metamodel to support the specification of some rules that have not been specified. The construction of MDE frameworks is recognized to be an iterative process, in which the first step is building a deep understanding of a small part of the reference domain [51]. A deeper discussion on our plans for a comprehensive evaluation of the proposed approach are reported in the next section.

### 7.1.4 Threats to Validity

The evaluation presented in this section is subject to some threats to validity, mainly originating from the challenges described in Section 4.3. We discuss here the most important ones, together with our discussion on how they could be mitigated.

Perhaps the most important threat to validity is the correlation existing between the definition of the metamodel and its application for the specification of coding rules. In fact, both activities were performed by the author of this dissertation.

The second threat is the subjectiveness in judging whether a certain CCSL specification correctly represents the original rule written in natural language. As discussed in Section 4.3 this threat is difficult to mitigate, given the ambiguity of the natural language. A possible way to mitigate this problem is to perform an extensive evaluation with experts, or in general with external participants. Two kinds of evaluation can be performed: i) provide them with CCSL specifications and ask them to assess whether the specifications correctly represent the corresponding textual rule, or ii) provide them with a set of textual rules and ask them to devise an equivalent CCSL specification. The second experiment would be ideal, but it is considerably more difficult to realize, because of the high demands posed on the participants, in terms of required time and skills.

A more systematic way to mitigate the subjectiveness problem is to actually generate checkers from the modeled rules, run them on real source code, and then compare the results with those obtained with existing static analysis tools. In this context, we concretely implemented a set of transformations to derive concrete checkers from the modeled rules. The next experiment (Checker Generator Evaluation) presents an experiment in this direction.

## 7.2 Checker Generator Evaluation

### 7.2.1 Methodology

As seen in previous chapters, we have designed a set of transformations to derive concrete checkers from CCSL specifications (OCL queries). To evaluate the generated checkers we compared their results with those given by PMD. We discuss the comparison between CCSL and PMD tools through a Venn diagram, as illustrated in Figure 7.1.

Given a certain coding rule, the "Real Violations" is the set of real violations that should be reported by the tools. The "CCSL Violations" (blue circle) represents the violations reported by the CCSL framework. Ideally, this is the same as "Real Violations", but in reality it may contains false positives and false negatives. Finally, the "PMD Violations" represents the violations reported by the PMD tool. Note, however, that since we do not know the ground truth ("Real Violations"), our objective is to compare our tool with an existing tool and see if it performs similarly or maybe better.

As can be noted in the Venn diagram, there a total of seven distinct areas. Each area is described below:

**A**. This area includes the CCSL violations that are true positives, as well as the false negatives of the PMD tool.

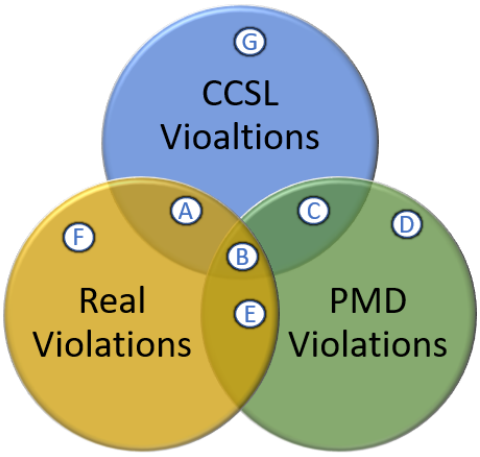



Figure 7.1: Venn diagram to represent the comparison between the CCSL and PMD tools

**B**. This areas includes the violations reported by booth CCSL and PMD reports that are true positive.

**C**. This areas includes the violations reported by booth CCSL and PMD reports that are false positives.

**D**. This areas includes the violations reported by PMD only that are false positives.

**E**. This area includes the violations reported by PMD only that are true positive, i.e., the false negatives of the CCSL framework.

**F**. This area includes the violations that are not reported by booth CCSL framework and PMD tool.

**G**. This areas includes the violations reported by the CCSL framework that are false positives.

This experiment is divided into two parts. In the first part, we assigned a classification for each code rule depending on the comparison between the outputs from CCSL and PMD. For each rule verified on a certain project: $V_{CCSL}$ is the set of violations reported by CCSL; $V_{PMD}$ is the set of violations reported by PMD. Sets A ...F are as in the Figure 7.1. Therefore, a rule can be classified in one of the following seven categories:

- Exact: The CCSL reports are exatly the same of PMD reports (Equation 7.1).

$$V_{CCSL} = V_{PMD} \ \wedge (V_{CCSL} \cup V_{PMD}) \neq \emptyset \tag{7.1}$$

- CCSL$^{+}$: CCSL performed better than PMD (Equation 7.2). There are differences between the CCSL and PMD reports. By analyzing these differences manually and taking the PMD rule documentation as a reference, one or both of the following scenarios occurred: i) the violations reported by CCSL only are all true-positive; ii) the violations reported by PMD only are all false-positive.

$$(G \cup E) = \emptyset \ \wedge (A \cup D) \neq \emptyset \tag{7.2}$$

- PMD$^+$: PMD perfomed better than CCSL (Equation 7.3). There are differences between the CCSL and PMD reports. By analyzing these differences manually and taking the PMD rule documentation as a reference, one or both of the following scenarios occurred: i) the violations reported by CCSL only are all false-positive; ii) the violations reported by PMD only are all true-positive.

$$(A \cup D) = \emptyset \ \wedge (E \cup G) \neq \emptyset \tag{7.3}$$

- Partial: There are differences between the CCSL and PMD reports. By analyzing these differences manually and taking the PMD rule documentation as a reference, we found out that there is at least one CCSL true-positive report that PMD does not have reported as well as there is at least one PMD true-positive report that CCSL does not have included.

$$A \neq \emptyset \wedge E \neq \emptyset \tag{7.4}$$

- NoViolations: Both CCSL and PMD tools did not report any alert (Equation 7.5).

$$V_{CCSL} = V_{PMD} = \emptyset \tag{7.5}$$

- NoSpecification: It was not possible to specify the rule with CCSL.

It is important to note that we do not have any pre-established oracle to define whether the CCSL report or PMD report is a false-positive, false-negative, etc. The goal of this experiment is to compare how similar the CCSL reports are against the PMD reports. In this context, if a CCSL reports is exactly the same of PMD reports (exact classification), this does not mean that there are no false positives or false negatives reported. In case some of the CCSL reports differ from PMD reports and vice-versa, then we manually check the difference between the reports and we classify our result based on the documentation of the PMD rule giving one of the above classification mentioned (CCSL$^+$, or PMD$^+$, or Partial).

When the first part of the experiment is completed (i.e., when the code rules were classified per project), then we proceed to the second part of the experiment, i.e., to assign a final classification of the code rules by joining the results of all projects. By excluding the classifications *NoViolations* and *NoSpecification*, we assigned a final classification to a code rule according to the following criteria.

- Equal: For all the projects the rule is classified as *Exact*.
- Better: For each project the rule is either classified as *Exact* or *CCSL$^+$*, where there is at least one *CCSL$^+$*.
- Worse: For each project the rule is either classified as *Exact* or *PMD$^+$*, where there is at least one *PMD$^+$*.
- Inconcluding: All the remaining cases.

### 7.2.2 Selected Coding Rules

The transformations to derive concrete checkers from CCSL specifications do not cover all metaclasses defined in CCSL. In fact, the transformations developed here are a prototype to collect evidence that it is possible to concretely derive checkers to validate a code rule specified in CCSL. For this reason the number of selected rules is samller than in the previous experiment.

To decide whether a PMD coding rule should be selected or not we first executed all the PMD rules of the previous experiment on the target systems. We then selected all those rules that reported at least one violations in any of the target systems. We note that this is somehow an advantage to PMD, as we potentially excluded rules in which it would experience false negatives only.

As a result, we selected 53 rules from a total of 138 rules (38%).

### 7.2.3 Target Systems

The selected projects are listed in Table 7.2, including their names (linked to their github repository), selected versions, and a short description.

Table 7.2: Software projects selected for the evaluation.

| *Name* | *Commit Code* | *Description* |
|---|---|---|
| WebGoat | edd6b7d | A deliberately insecure web application maintained by OWASP designed to teach web application security lessons. |
| TeaStore | 5cab414 (tag v1.3.4) | A micro-service reference and test application to be used in benchmarks and tests. |
| WSVD-Bench | 95f08bb | The source code of the services that serve as workload of a benchmark for tools able to detect SQL injection vulnerabilities in web services. |

### 7.2.4 Results

Results of the analysis are reported in Table 7.3. All the outputs from CCSL and PMD tools are available on GitHub, as well as the derived OCL queries from the CCSL specifications.

Table 7.3: Classification of rules per each system

| **System** | **Classification** | | | | | | **Total** |
|---|---|---|---|---|---|---|---|
| | *CCSL*$^+$ | *PMD*$^+$ | *Exact* | *Partial* | *NoViolations* | *NoSpecification* | |
| Tea Store | 7 | 6 | 20 | 0 | 17 | 3 | 53 |
| Web Goat | 9 | 4 | 26 | 0 | 11 | 3 | 53 |
| WSVD | 1 | 1 | 18 | 2 | 28 | 3 | 53 |

During the experiment we generated 50 checkers from a total of 53 specifications (94%). Even though the transformations is still a prototype, we believe that results confirm that a MDE approach using OCL queries is a feasible approach when compared to an existing static analysis tool such as PMD. The Table 7.3 includes the classification “NoViolations”,

i.e., for a given rule $r$ in a project $p$, no violations were reported either by PMD or CCSL. In order to focus on rules that resulted in violations, the Table 7.4 presents the results excluding the classification "NoViolations".

Table 7.4: Classifications of rules per system excluding "NoViolations"

| **System** | **Classification** | | | | | **Total** |
|---|---|---|---|---|---|---|
| | *CCSL*$^+$ | *PMD*$^+$ | *Exact* | *Partial* | *NoSpecification* | |
| Tea Store | 7 (19%) | 6 (17%) | 20 (56%) | 0 | 3 (8%) | 36 |
| Web Goat | 9 (21%) | 4 (10%) | 26 (62%) | 0 | 3 (7%) | 42 |
| WSVD | 1 (4%) | 1 (4%) | 18 (72%) | 2 (8%) | 3 (12%) | 25 |
| **Average** | 15% | 10% | 63% | 3% | 9% | |

Most of the rules were classified as *Exact* (63% on average) and *CCSL*$^+$ (15% on average), meaning that we achieved the same or even better results of a popular existing static analysis tool. On the other hand, there are also cases where we could not specify a rule using our language (3 rules out of 53 in total) as well as there are cases where the PMD implementation has achieved better results than our approach (10% on average).

In order to understand whether classifications are consistent across the target projects, i.e., if a rule is classified as exact in one project, it should also be classified as exact in other projects, we have joined the results of each project, giving a final classification of a rule according to the criterias established in the Methodology Section. The Table 7.5 reports the final classifications of each selected rule.

Table 7.5: Final Classifications

| *Better* | *Worse* | *Equals* | *Inconcluding* | *NoSpecification* | **Total** |
|---|---|---|---|---|---|
| 10 (19%) | 5 (9%) | 32 (60%) | 3 (6%) | 3 (6%) | 53 |

The results show that most of rules classified as *Exact* in one project are also classified as *Exact* (60%) or even *CCSL*$^+$ (19%) in another projects. We registered only five rules (9%) where the PMD tool has performed better across the projects. There are three rules classified as *Inconcluding* which means that we can not argument which tool performed better since the result was not stable between the projects, i.e., in some projects the rule was classified as *CCSL*$^+$ and others as *PMD*$^+$.

### 7.2.5 Threats to Validity

Similarly to the previous experiment, this experiment is also subject to some threats to validity we would like to discuss.

First, there is a subjectiveness in judging whether an alert reported by CCSL or PMD is a true positive or false positive. To avoid mistakes, we take as reference the documentation of PMD rules. However, as mentioned before, sometimes the documentation is quite simple and we need to execute some PMD rules in the projects or check its implementation to see what is the real behavior of the rule. This can possibly lead to a misunderstanding

of the rule. We plan to mitigate this threat by making other persons to classify the rules, since only the author of this dissertation has classified the rules.

Second, the rules selected in this experiment as well as the target projects can be considered as a small set. However, the transformations is still a prototype. As the transformations becomes more stable, we plan to increase the number of rules to be selected as well as to consider another static analysis tools.

# Chapter 8

# Final Remarks

## 8.1 Conclusion

In this study we proposed an approach to provide structured specifications of coding conventions, by applying model-driven engineering techniques. To the best of our knowledge, there is little work in such direction. We defined a language, CCSL, that can be used to specify coding rules in a structured way. We also defined model transformations to derive OCL queries from the CCSL specification. Such queries are applied in a Java model, which is extracted from the source code thanks to the MoDisco tool. The result of the derived OCL queries are the Java elements that are violations of the rule specified in CCSL language.

In order to evaluate the proposed approach, we performed two experiments. The first experiment aimed to evaluate the Coding Conventions Specification Language metamodel, i.e., to check whether the current CCSL metamodel is able to specify real coding conventions. We analyzed a total of 216 individual rules from two large sets of existing coding conventions. Results are promising, but also show that the focus of rules and the way they are written have a fundamental impact. Overall, it was possible to represent 63% of the considered coding rules, which can be considered satisfactory for a first investigation.

The second experiment aimed to evaluate the derived OCL queries, i.e., their capability to find violations of the specified rule. In this experiment we compared our results to the results of the PMD tool, a popular static code analysis tool. We selected a total of 53 rules from those implemented in the PMD tool and we ran such rules in three real open source systems. For each rule one of the followings classification has been assigned: *Better* (CCSL perfomed better than PMD), *Worse* (CCSL perfomed worse than PMD), *Equal* (CCSL results are exatcly the same as in *PMD*), *Inconcluding* (we can not argue which tool performed better), and *NoSpecification* (it was not possible to specify a PMD rule using CCSL). In general, we achieve equal or better results of the PMD tool in more than half of the selected rules (79%), while only 6% of the rules could not be specified using our language. There are also cases where PMD performed better than our approach (9%) as well as cases where the results were different for each of the tools (6%). We consider the results as satisfactory for a first investigation, since CCSL is a Domain-Specific Language and naturally it imposes some restrictions in terms of specifying any kind of rules. In the other hand, PMD rules can be implemented using the Java language (a general purpose

language), i.e., there is no restriction in terms of the kind of rules that can be checked.

## 8.2 Using CCSL for Fault Injection

As mentioned in Section 1.2 the author of this dissertation made an exchange in the UNIFI. The original purpose of CCSL is to specify rules of coding conventions, and then generate checkers that are able to find violations of such rules in the source code. In this exchange, we have extended the CCSL where faults can be specified, i.e., we have extended the CCSL where it is possible to specify to: i) which elements in the source code are selected; and ii) which actions (replace, delete, etc.) should be performed in these selected elements in order to inject a fault in the software. Once the specification is done, a model-transformation automatically generates an implementation of the needed Injector(s). Given any software in the target programming language (the Java language), the injector is able to i) find the possible injection points and then ii) concretely inject the faults in the identified injection points.

We have specified 13 commons fault types and we applied them to six Java projects that belong to different domains. In general, our tool was able to find 61,061 injection points and successfully inject 49,304 faults (80,7%), showing the promising use of model-driven engineering to inject fault types. This work has been submitted to the 31st International Symposium on Software Reliability Engineering (ISSRE) 2020.

## 8.3 Future Work

This study relies on the extraction of Java model by the MoDisco tool. However, the MoDisco tool was designed to work as a Eclipse-Plugin, and thus our current implementation is bound to the Eclipse IDE. In addition to that, the MoDisco Java metamodel is not updated to the current Java version (e.g., it does not support lambda expression). Therefore, it is necessary to create or update the Java metamodel and to extract it without any dependency to IDEs.

We also can work on refining and extending the CCSL metamodel, aiming to cover a wider range of rules. It is also possible to create a concrete textual syntax to the CCSL metamodel, making it more flexible in the development process. Another interesting direction to investigate is to derive guidelines for the definition of coding conventions, that is, understanding which characteristics make them suitable for a structured specification and automated verification.

Finally, as the CCSL framework is being refined, a more extensive evaluation should be designed.

## 8.4 Acknowledgment

This work has been supported by the São Paulo Research Foundation (FAPESP) with grants #2018/11129-8 and #2019/06799-7 (BEPE scholarship). This work has been

developed in the context of the H2020-MSCA-RISE-2018 “ADVANCE” project (grant 823788).

# Bibliography

[1] Acceleo. https://www.eclipse.org/acceleo/. (Accessed October 24, 2019).

[2] CheckStyle. http://checkstyle.sourceforge.net/ (Accessed October 24, 2019).

[3] Gmf. https://wiki.eclipse.org/Graphical_Modeling_Framework.

[4] PMD. https://pmd.github.io/ (Accessed October 24, 2019).

[5] QJ Pro. http://qjpro.sourceforge.net/ (Accessed October 24, 2019).

[6] Sirius. https://www.eclipse.org/sirius/.

[7] Xtext. https://www.eclipse.org/Xtext/documentation/308_emf_integration.html.

[8] Improving security using extensible lightweight static analysis. 19(1):42–51, 2002.

[9] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[10] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[11] ATLAS group LINA & INRIA Nantes. ATL: Atlas Transformation Language – ATL Starter's Guide. Version 0.1, December 2005.

[12] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56, pages 2:1–2:25, Dagstuhl, Germany, 2016.

[13] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 173–174, 2010.

[14] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, (4):171–188, 2005.

[15] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[17] Boryana Goncharenko and Vadim Zaytsev. Language design and implementation for the domain of coding conventions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*, pages 90–104, 2016.

[18] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591, 2015.

[19] G. J. Holzmann. The power of 10: rules for developing safety-critical code. *IEEE Computer*, 39(6):95–99, 2006.

[20] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, December 2004.

[21] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 471–480, May 2011.

[22] INCOSE (International Council on Systems Engineering). A World in Motion – Systems Engineering Vision 2025, 2014.

[23] Jet Propulsion Laboratory (JPL). JPL Java Coding Standard – JPL Institutional Coding Standard for the Java Programming Language. Version 1.0, March 2014.

[24] Tae-Woong Kim, Tae-Gong Kim, and Jai-Hyun Seu. Specification and automated detection of code smells using ocl. *International Journal of Software Engineering and Its Applications*, 7(4):35–44, 2013.

[25] Panagiotis Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[26] MIRA Limited. Guidelines for the use of the C language in critical systems. MISRA-C:2004, October 2004.

[27] MIRA Limited. Guidelines for the use of the C++ language in critical systems. MISRA-C++:2008, June 2008.

[28] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[29] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[30] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3-4):345–361, 2010.

[31] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, Sep. 2018.

[32] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*, 101(2):161–185, February 2019.

[33] Object Management Group. Model Driven Architecture (MDA) – MDA Guide rev. 2.0. ormsc/2014-06-01, June 2014.

[34] Object Management Group. Object Constraint Language. formal/2014-02-03, February 2014. `https://www.omg.org/spec/OCL/`.

[35] Object Management Group. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM). Version 1.4, formal/16-09-01, September 2016.

[36] Object Management Group. MOF model to text transformation. formal/2014-02-03, January 2016. `https://www.omg.org/spec/MOFM2T/About-MOFM2T/`.

[37] Object Management Group. OMG Meta Object Facility (MOF) Core Specification. formal/2016-11-01, November 2016. Version 2.5.1.

[38] Object Management Group. OMG Unified Modeling Language (OMG UML). formal/2017-12-06, December 2017. Version 2.5.1.

[39] Miroslaw Ochodek, Regina Hebig, Wilhelm Meding, Gert Frost, and Miroslaw Staron. Recognizing lines of code violating company-specific coding guidelines using machine learning. *Empirical Software Engineering*, 25(1):220–265, nov 2019.

[40] Claudia Raibulet, Francesca Arcelli Fontana, and Marco Zanoni. Model-driven reverse engineering approaches: A systematic literature review. *IEEE Access*, 5:14516–14542, 2017.

[41] Elder Rodrigues Jr. and Leonardo Montecchi. CCSL Metamodel. `https://github.com/Elderjr/Coding-Conventions-Specification-Language` (Accessed October 24, 2019).

[42] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[43] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 504–507, 2011.

[44] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia. Maintainability and source code conventions: An analysis of open source projects. Technical Report TR11-06, University of Alberta, Department of Computing Science, June 2011.

[45] Warren B. Smith. Characteristics of Model-Based Systems Engineering. 2013 INCOSE Chapter Meeting, 2013.

[46] Software Engineering Institute – Carnegie Mellon University. SEI CERT Coding Standard. `https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards/` (Accessed October 24, 2019).

[47] Ian Sommerville. *Software engineering*. New York: Addison-Wesley, 2010.

[48] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, December 2008.

[49] Ye Henry Tian. String concatenation optimization on java bytecode. In *Proceedings of the International Conference on Software Engineering Research and Practice, SERP 2006, Volume 2*, pages 945–951, Las Vegas, Nevada, USA, 2006.

[50] G. H. Travassos. Software defects: Stay away from them. do inspections! In *2014 9th International Conference on the Quality of Information and Communications Technology*, pages 1–7, 2014.

[51] Markus Voelter. Best Practices for DSLs and Model-Driven Development. *Journal of Object Technology*, 8(6), 2009.

[52] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, January 2013.

[53] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. In *IFIP International Conference on Testing of Communicating Systems*, pages 40–55, 2005.

[54] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[55] World Wide Web Consortium (W3C). XML Path Language (XPath) 3.1. W3C Recommendation, March 2017.

[56] Yan Wu, Robin A. Gandhi, and Harvey Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems (SESS 2010)*, pages 22–28, Cape Town, South Africa, 2010. ACM.